Lightweight Structure in Text

Robert C. Miller

May 2002
CMU-CS-02-134
CMU-HCII-02-103

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the National Science Foundation under grant no. IRI-9319969 and grant no. IIS-0117658, the Army Research Office (ARO) under National Defense Science and Engineering Grant no. DAAH04-95-1-0552, the Defense Advanced Research Project Agency (DARPA) - US Army under contract no. DAAD1799C0061, DARPA - Navy under contract no. N66001-94-C-6037, N66001-96-C-8506, and the USENIX Association.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **MAY 2002**	2. REPORT TYPE	3. DATES COVERED **00-05-2002 to 00-05-2002**
4. TITLE AND SUBTITLE **Lightweight Structure in Text**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**341**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Keywords: text processing, structured text, pattern matching, regular expressions, grammars, web automation, repetitive text editing, machine learning, programming-by-demonstration, simultaneous editing, outlier finding, LAPIS

Abstract

Pattern matching is heavily used for searching, filtering, and transforming text, but existing pattern languages offer few opportunities for reuse. *Lightweight structure* is a new approach that solves the reuse problem. Lightweight structure has three parts: a model of text structure as contiguous segments of text, or *regions*; an extensible library of structure abstractions (e.g., HTML elements, Java expressions, or English sentences) that can be implemented by any kind of pattern or parser; and a *region algebra* for composing and reusing structure abstractions. Lightweight structure does for text pattern matching what procedure abstraction does for programming, enabling construction of a reusable library.

Lightweight structure has been implemented in LAPIS, a web browser/text editor that demonstrates several novel techniques:

- *Text constraints* is a new pattern language for composing structure abstractions, based on the region algebra. Text constraint patterns are simple and high-level, and user studies have shown that users can generate and comprehend them.
- *Simultaneous editing* uses multiple selections for repetitive text editing. Multiple selections are inferred from examples given by the user, drawing on the lightweight structure library to make fast, accurate, domain-specific inferences from very few examples. In user studies, simultaneous editing required only 1.26 examples per selection, approaching the 1-example ideal.
- *Outlier finding* draws the user's attention to inconsistent selections or pattern matches — both possible false positives and possible false negatives. When integrated into simultaneous editing and tested in a user study, outlier finding reduced user errors.
- *Unix tools for structured text* extend tools like `grep` and `sort` with lightweight structure, and the *browser shell* integrates a Unix command prompt into a web browser, offering new ways to build pipelines and automate web browsing.

Theoretical contributions include a formal definition of the region algebra, data structures and algorithms for efficient implementation, and a characterization of the classes of languages recognized by algebra expressions.

Lightweight structure enables efficient composition and reuse of structure abstractions defined by various kinds of patterns and parsers, bringing improvements to pattern matching, text processing, web automation, repetitive text editing, inference of patterns from examples, and error detection.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

This thesis is dedicated to:

- my dear wife Laura, who puts up with a lot;
- my parents Larry and Marian, and my brothers Craig and Eric, for their love and support, and for only rarely asking when I was going to get a real job;
- Dick Kollin and Dave Fraser, whose shameless exploitation of teenage labor gave me my start in this crazy business;
- Charles Leiserson, who encouraged me to go on to grad school at CMU and predicted I'd come back someday;
- Brad Myers, who pointed me in the right direction, gave me the ball, and let me run with it;
- my other committee members David Garlan, Jim Morris, and Brian Kernighan;
- my fellow grad students James Landay, Rich McDaniel, John Pane, Jeff Nichols, and Jake Wobbrock;
- Sharon Burks and Ava Cruse, who made impossible things seem easy;
- Yuzo Fujishima, who kindly provided materials for my user studies;
- the wonderful people with whom I have shared offices (and laughter) over the years at CMU, including George Necula, Fay Chang, José Brustoloni, Adam Berger, Chris Colohan, Hakan Younes, Umair Shah, Bartosz Przydatek, and Sanjit Seshia;
- everyone who gave valuable feedback about LAPIS, including Rich Clingman, Monty Zukowski, Julián Jesús Martínez López, John Padula, Ben Bostwick, Franklin Chen, Chris Long, and John Gersh.

Chapter 1

Introduction

Computer users are surrounded by text — personal documents, web pages, email messages, newsgroup postings, program source code, data files, configuration files, event logs, and more. Even with the rise of graphical user interfaces, multimedia personal computers, and broadband networks with streaming video and audio, most of the information consumed and produced by computer users still comes in textual form.

Text is full of *structure* designed to help readers understand and use the information it contains. Figures 1.1–1.4 illustrate some of the structure found in email messages, web pages, and source code. Some structure is explicitly labeled, like the headers of the email message in Figure 1.1; some is more implicit, like the phone number and signature in the body of the message. Some structure is formal and hierarchical, with every part appearing in a fixed, well-defined place, like the Java language syntax in Figure 1.4; some is more informal, like the book titles and authors in Figure 1.2, or the advertisement in Figure 1.3.

Often a document has several distinct layers of structure. For example, at the highest level, Figure 1.2 is a list of books, with attributes like title, year, author, genre, and format. At a lower level, the page is laid out as a table, with rows, columns, header cells, and data cells. At a still lower level, not visible in the figure, the page is a tree of HTML markup elements, and at the lowest level, a string of ASCII characters. All these layers of structure provide useful information for understanding or processing the document.

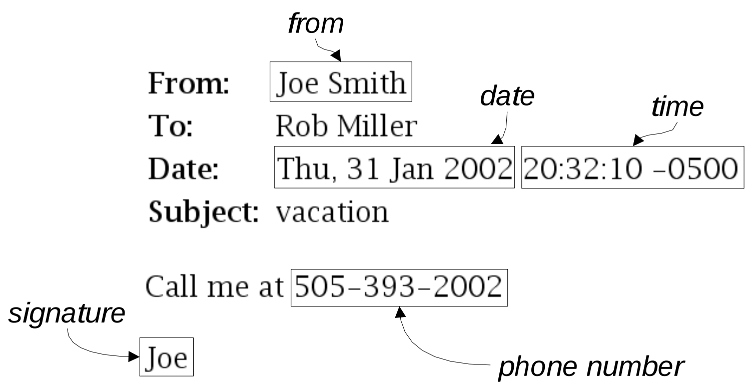
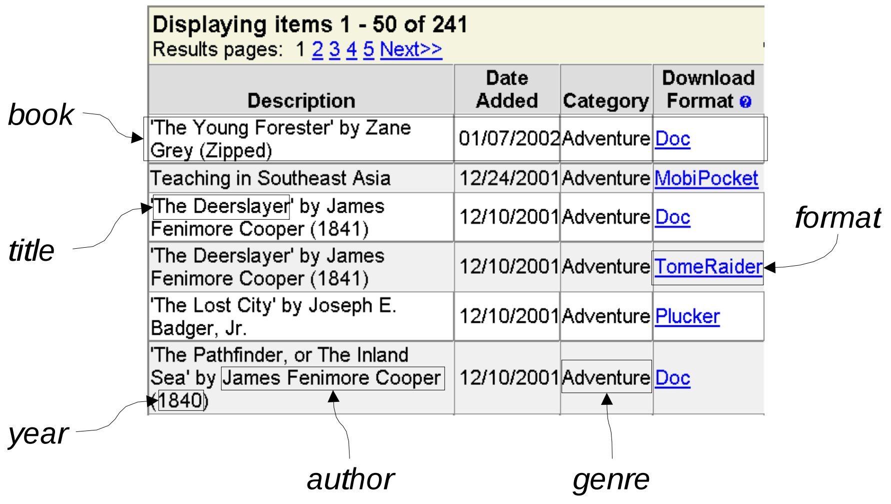


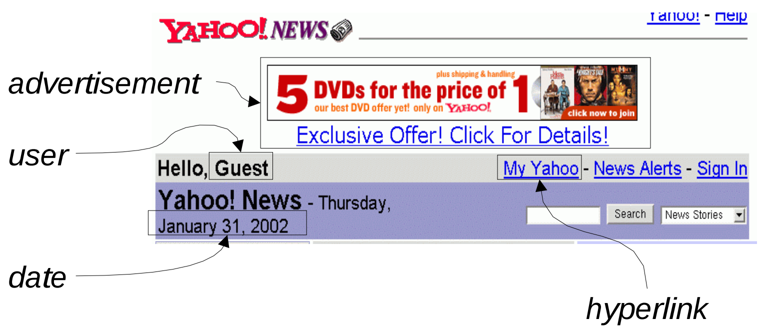
Figure 1.1: An email message with some of its structure labeled.



Displaying items 1 - 50 of 241
Results pages: 1 2 3 4 5 Next>>

Description	Date Added	Category	Download Format
'The Young Forester' by Zane Grey (Zipped)	01/07/2002	Adventure	Doc
Teaching in Southeast Asia	12/24/2001	Adventure	MobiPocket
'The Deerslayer' by James Fenimore Cooper (1841)	12/10/2001	Adventure	Doc
'The Deerslayer' by James Fenimore Cooper (1841)	12/10/2001	Adventure	TomeRaider
'The Lost City' by Joseph E. Badger, Jr.	12/10/2001	Adventure	Plucker
'The Pathfinder, or The Inland Sea' by James Fenimore Cooper (1840)	12/10/2001	Adventure	Doc

Figure 1.2: A web page that lists electronic books available for downloading.



Yahoo! - Help

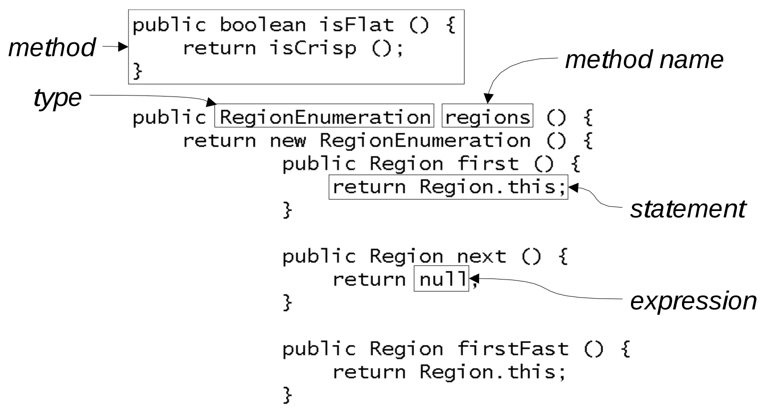
Exclusive Offer! Click For Details!

Hello, Guest My Yahoo - News Alerts - Sign In

Yahoo! News - Thursday, January 31, 2002

Search News Stories

Figure 1.3: A web page from Yahoo.



```
public boolean isFlat () {
    return isCrisp ();
}

public RegionEnumeration regions () {
    return new RegionEnumeration () {
            public Region first () {
                return Region.this;
            }

            public Region next () {
                return null;
            }

            public Region firstFast () {
                return Region.this;
            }
```

Figure 1.4: Part of a Java program.

Structure has value not only for reading and understanding text, but also for manipulating it — searching, editing, filtering, transforming, or rearranging its elements. An email user may want to find the message from Joe with his phone number in it. A book shopper may want to limit the books to a certain genre or format. A web surfer may want to strip advertisements from a web page. A programmer may want to find the methods that return `null` and edit them to throw an exception instead.

These manipulations require some way to describe the structure to be manipulated, such as books, advertisements, or Java methods.. Previous systems have generally chosen one of two well-known mechanisms for describing text structure: *grammars* or *regular expressions*. Grammars are well suited to describing formal, hierarchical structure imposed on an entire document, like programming language syntax or document markup. Grammars are used by syntax-directed program editors like Gandalf [HN86] and the Cornell Program Synthesizer [RT89], and by markup languages like SGML [Gol90] and XML [W3C00]. Regular expressions, on the other hand, are suitable for matching local or informal structure, like phone numbers or email addresses, but lack the expressive power to describe hierarchical structure. Regular expressions are used heavily by awk [AKW88] and Perl [WCS96].

One problem with previous approaches to structure description is that they provide little opportunity for *reuse*, especially between the two approaches. Perl programmers struggle to describe HTML elements with complex regular expressions, despite the fact that grammar-based HTML parsers already encapsulate that knowledge. A grammar-based XML tool cannot take apart an email address like `rcm@cs.cmu.edu` that appears in an XML document, because the parts of the address are not explicitly marked up. Yet a regular expression could handle this task easily.

This thesis describes a new approach to structure description that solves the problem of reuse: *lightweight structure*. Lightweight structure allows a system's basic structure concepts to be defined by a variety of mechanisms — grammars, regular expressions, or in fact any kind of pattern or parser. These structure concepts can then be composed and reused, regardless of the mechanism used to define them.

The lightweight structure approach has three parts:

1. a *model* of text structure as contiguous segments of text called *regions*. Each of the labeled rectangles in Figures 1.1–1.4 is a region.

2. an extensible *library* of structure abstractions. A structure abstraction is a named concept, like a word, a sentence, or a phone number. Every label in Figures 1.1–1.4 is a plausible structure abstraction. A structure abstraction can be defined by any kind of parser or pattern. The output of a structure abstraction is the set of all regions in the text that belong to the concept — e.g., the set of words, the set of sentences, or the set of Java expressions.

3. an *algebra* for combining sets of regions, based on relations like *before, after, in,* and *contains*. The algebra allows structure abstractions to be composed to create new abstractions.

The ability to compose and reuse structure abstractions is the most powerful aspect of lightweight structure. Suppose the library includes a hyperlink abstraction defined by an HTML parser, and an email address abstraction defined by a regular expression. Lightweight structure allows these abstractions to be composed to find hyperlinks that contain email addresses, without modifying either of the constituent abstractions. The resulting composition, which might be called an email link, can be put back in the library as a new abstraction.

Lightweight structure does for text pattern matching what procedural abstraction does for programming. It provides a uniform interface, analogous to a procedure calling convention, that enables the creation of genuine abstractions with encapsulation and information hiding. Abstractions need not be shoehorned into one particular description language with limited expressive power. Instead, a structure abstraction can be expressed in any kind of pattern language, including context-free grammars, regular expressions, and region algebra expressions. Alternatively, an abstraction can be implemented by some process, such as a hand-coded scanner, a Turing machine, or even a human being doing manual selection with a mouse. Different ways of implementing a structure abstraction may have different properties in terms of performance, cost, or robustness, but functionally they are the same. To put it concretely, a user can use the hyperlink abstraction in a pattern without knowing how it is implemented, in the same way that a C programmer can call `strcpy` without knowing the details of its implementation. Lightweight structure enables the construction of a reusable text pattern library analogous to a function library.

Lightweight structure has the added advantage that the abstractions in the library are available at all times. Abstractions like phone numbers, URLs, and sentences can be used regardless of whether the user is looking at a web page, an email message, or a Java program.

1.1 Applications

Lightweight structure has a wide range of possible applications, a number of which are explored in this dissertation:

- **Pattern matching.** The region algebra serves as a basis for a new pattern language, called *text constraints* (TC), that permits the composition and reuse of lightweight structure abstractions. Because TC can use structure abstractions as primitives, it is higher-level than pattern languages that refer only to characters, such as regular expressions and grammars. As as result, TC patterns tend to be simpler, and easier to read and write. A user study showed that users can successfully read and write TC patterns.

- **Unix-style text processing.** A cornerstone of the Unix environment is its basket of generic tools, like `grep` and `sort`, that can be combined into pipelines and scripts to solve text-processing problems without writing a custom program. Unfortunately, these tools can be hard to apply to richly structured text like web pages, source code, and structured data files, because the tools generally assume that the input is an array of lines. Lightweight structure allows this limitation to be overcome, so that generic Unix-like tools can be used to filter and manipulate other kinds of structure, like the books in Figure 1.2 or the Java methods in Figure 1.4.

- **Web automation.** More and more information comes to users from the World Wide Web, and more and more work is done by interacting with services through the Web rather than running applications locally. Lightweight structure makes it easier for a user to script web interactions — clicking on hyperlinks, filling in and submitting forms, and extracting structured data from web pages — so that repetitive browsing activities can be automated, and web-based services and information sources can be incorporated into other programs.

- **Repetitive text editing.** Text editing is full of repetitive tasks. The region set model of lightweight structure provides a new way of performing these tasks: *multiple-selection editing*. Multiple-selection editing allows the user to edit multiple regions in a document at the same time, with the same commands as single-selection editing. Selections can be made by the mouse, by lightweight structure abstractions, or by writing a pattern. Multiple-selection editing is more interactive than other approaches to repetitive editing, like keyboard macros or find-and-replace, and lightweight structure makes it easier to describe the desired selections.

- **Inferring patterns from examples.** Many applications of text pattern matching can be improved by learning the patterns from examples, among them information extraction [KWD97, Fre98] and repetitive text editing by demonstration [WM93, Mau94, Fuj98, LWDW01]. Previous systems learned from fixed, low-level structure concepts, like words and numbers. Lightweight structure provides a library of high-level concepts, so inferences can directly refer to HTML or Java syntax without having to learn it first. This advantage is exploited by two techniques described in this thesis. The first is an algorithm that infers TC patterns from positive and negative examples. The inferred patterns can be used for any of the applications described previously, including Unix-style text processing, web automation, and adding more abstractions to the library. The second technique, called *simultaneous editing*, is expressly designed for repetitive text editing. Simultaneous editing is multiple-selection editing where the multiple selections are inferred from examples, using not only lightweight structure but also a special heuristic for repetitive editing to make very accurate inferences with very few examples. Simultaneous editing has been found to be fast and usable by novices, closely approaching the ideal of 1 example per inference.

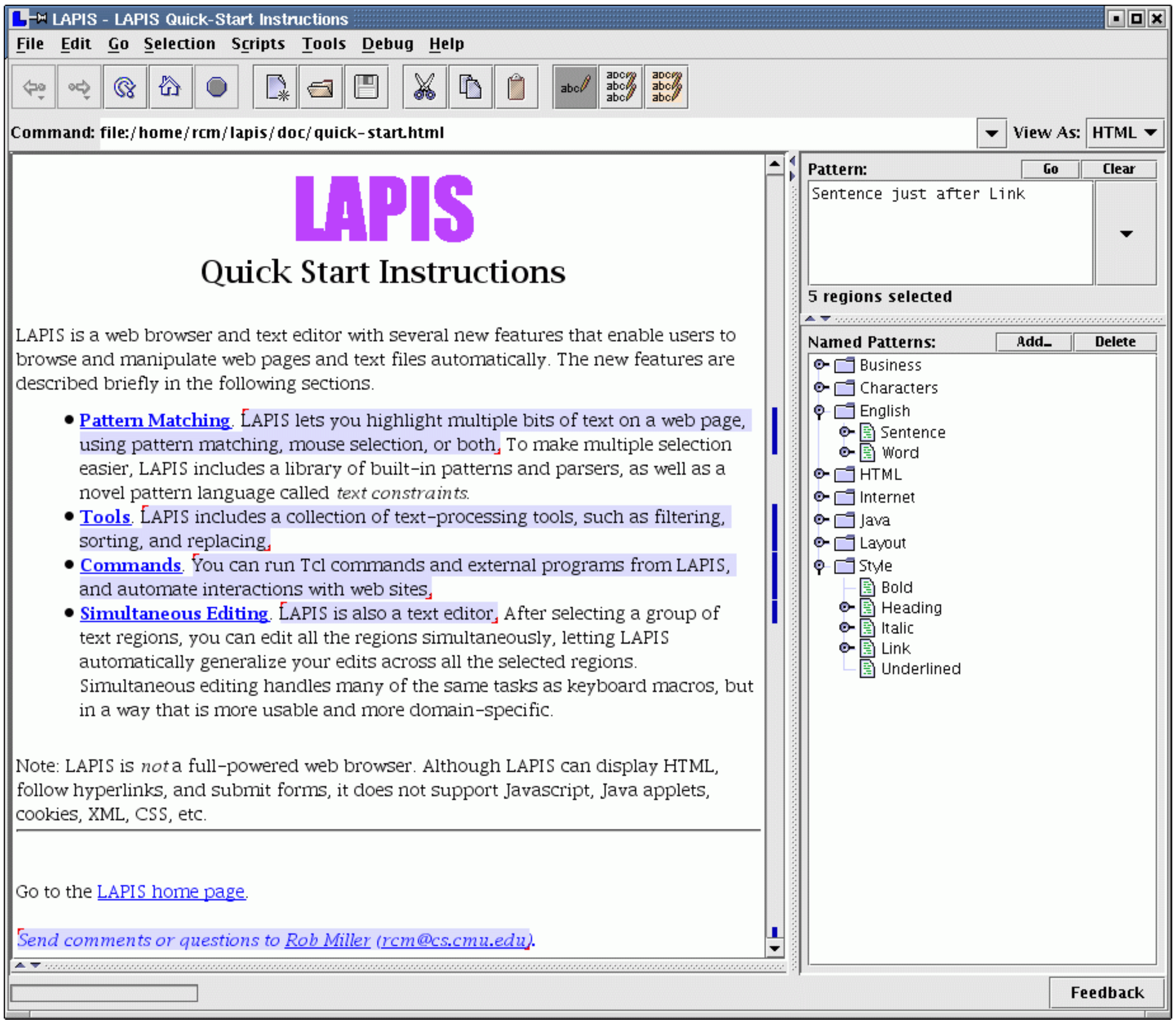
- **Error detection.** *Outlier finding* is a new way to reduce errors by drawing the user's attention to inconsistent lightweight structure. Outlier finding can point out both possible false positives and possible false negatives in a pattern match or multiple selection. When outlier finding was integrated into simultaneous editing, it was found to reduce user errors.

1.2 LAPIS

These applications have been implemented in a system called LAPIS,[1] a web browser and text editor that supports lightweight structure. A screenshot of LAPIS is shown in Figure 1.5. The major new features of LAPIS are:

- **Multiple selections.** Multiple regions of text can be selected at the same time, using pattern matching, inference, or manual selection with the mouse. Multiple selections can be used to extract, filter, sort, replace, and edit text in a document.

[1]LAPIS stands for Lightweight Architecture for Processing Information Structure.



Command: file:/home/rcm/lapis/doc/quick-start.html

Sentence just after Link

5 regions selected

LAPIS

Quick Start Instructions

LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel pattern language called *text constraints*.
- **Tools**. LAPIS includes a collection of text-processing tools, such as filtering, sorting, and replacing.
- **Commands**. You can run Tcl commands and external programs from LAPIS, and automate interactions with web sites.
- **Simultaneous Editing**. LAPIS is also a text editor. After selecting a group of text regions, you can edit all the regions simultaneously, letting LAPIS automatically generalize your edits across all the selected regions. Simultaneous editing handles many of the same tasks as keyboard macros, but in a way that is more usable and more domain-specific.

Note: LAPIS is *not* a full-powered web browser. Although LAPIS can display HTML, follow hyperlinks, and submit forms, it does not support Javascript, Java applets, cookies, XML, CSS, etc.

Go to the LAPIS home page.

Send comments or questions to Rob Miller (rcm@cs.cmu.edu).

Figure 1.5: The LAPIS web browser/text editor, which demonstrates how lightweight structure can be used in a text-processing system.

- **Structure library.** LAPIS includes an extensible library of structure abstractions, shown on the right side of Figure 1.5 under the heading "Named Patterns." The default library includes HTML syntax defined by an HTML parser, Java syntax defined by a Java parser, and a variety of concepts like words, sentences, lines, phone numbers, and email addresses, defined by regular expressions and TC patterns. The library can be extended by plugging in new parsers, writing patterns, inferring patterns from examples, or making selections manually with the mouse.

- **TC pattern language.** The TC pattern language allows abstractions from the library to be composed and reused. Figure 1.5 shows an example of a TC pattern, `Sentence just after Link`. TC patterns are used to make multiple selections, add abstractions to the structure library, specify the arguments of script commands, and display feedback about inferences.

- **Scripting language.** LAPIS includes a scripting language, based on Tcl, that can be used to write text-processing scripts using the structure library and TC patterns.

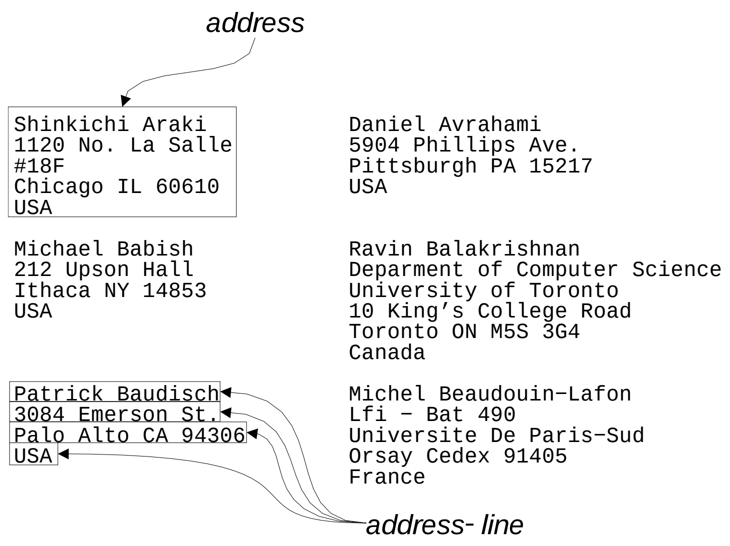
- **Browser shell.** Script commands can also be invoked interactively, using a novel user interface called a *browser shell*. The browser shell integrates the command interpreter into a web browsing interface, using the LAPIS command bar to enter commands, the browser pane to display output, and the browsing history to manage the history of outputs. External command-line programs can also be invoked through this interface, allowing existing Unix tools to be interleaved with LAPIS script commands. The browser shell also offers a new way to construct Unix-style pipelines, described in more detail in Chapter 8.

- **Inference.** LAPIS can infer patterns from positive and negative examples given with the mouse. Inferences are displayed both as multiple selections and as TC patterns. Inferred patterns can be used for editing or invoking commands, or placed into the structure library as new abstractions.

- **Outlier finding.** When LAPIS is inferring patterns from examples, it uses outlier finding to highlight unusual matches to the inferred pattern, so that the user can check them for possible errors. The outlier finder can also be invoked directly by the user to find possible errors in *any* multiple selection, including selections made by library abstractions or patterns written by the user.

The target audience for LAPIS, and indeed for the entire lightweight structure approach, covers a wide spectrum of computer users. Programmers can plug in a parser for their favorite programming language or data format, and then use it with any of the features of LAPIS: pattern matching, multiple-selection editing, scripting, even inference and error detection. Power users who are not necessarily programmers can describe structure by writing TC patterns or inferring them from examples, and then manipulate the structure with Unix-style text-processing commands and multiple-selection editing. Even casual or novice users can benefit from LAPIS: the user studies described in this thesis show that users can understand and use simultaneous editing and outlier finding without learning anything about lightweight structure.

The current LAPIS implementation is targeted primarily at three domains: HTML web pages, Java source code, and plain text. The examples in this dissertation are drawn from these domains.



```
Shinkichi Araki         Daniel Avrahami
1120 No. La Salle       5904 Phillips Ave.
#18F                    Pittsburgh PA 15217
Chicago IL 60610        USA
USA

Michael Babish          Ravin Balakrishnan
212 Upson Hall          Deparment of Computer Science
Ithaca NY 14853         University of Toronto
USA                     10 King's College Road
                        Toronto ON M5S 3G4
                        Canada

Patrick Baudisch        Michel Beaudouin-Lafon
3084 Emerson St.        Lfi - Bat 490
Palo Alto CA 94306      Universite De Paris-Sud
USA                     Orsay Cedex 91405
                        France
```

Figure 1.6: Not all useful abstractions can be represented by a contiguous region. In this plain text file laid out in two columns, each *address* actually consists of several noncontiguous regions. The closest we can get in the lightweight structure model is *address-line*.

Other text formats can also be processed, as long as their structure can be described by parsers, regular expressions, or TC patterns. LAPIS is not particularly useful for processing binary formats like Microsoft Word documents, but neither are Perl or awk. However, nothing precludes incorporating aspects of the lightweight structure approach, or particular techniques like simultaneous editing and outlier finding, into a monolithic word-processing application like Microsoft Word.

1.3 Limitations

The lightweight structure approach is designed for recognizing and exploiting structure, not creating or validating it. For example, a phone number abstraction can recognize phone numbers, but it cannot produce a template of a phone number for the user to fill in, or guarantee that all phone numbers in a document are formatted the same way. The grammar-based approach used by syntax-directed editors and SGML/XML is better suited to these tasks.

The model of structure as contiguous regions in a one-dimensional string can capture most kinds of text structure, but not all. In particular, some aspects of two-dimensional layout are impossible to represent with contiguous one-dimensional regions. Figure 1.6 shows a plain text file of postal addresses laid out as two columns. In this format, an address is not a single contiguous region in the file. Instead, each address is split across multiple lines, and the lines of one address are interleaved with the lines of another. There is no way to define a structure abstraction that returns each address as a single unit, corresponding to the *address* label shown in the figure. The closest we can get is the *address-line* abstraction shown in the figure, which returns the individual lines of the addresses but doesn't group them together. A similar problem is encountered with the *column* concept in HTML tables, since HTML specifies tables in row-major order. Although the model described in this thesis cannot represent these cases, it may be possible to extend the model; more will be said about this possibility in the conclusion. Regular expressions and grammars cannot express the *address* abstraction either, for the same reasons.

Like regular expressions and context-free grammars, the region algebra has limits on the classes of languages that it can recognize. The recognition power of the region algebra is not fixed, however, but rather depends on the power of the abstractions being composed. For example, algebra expressions over regular abstractions can recognize only regular languages, but algebra expressions over context-free abstractions can recognize *more* than just context-free languages. These results are proved in Chapter 5.

1.4 Contributions

My thesis statement is:

> Lightweight structure enables efficient composition and reuse of structure abstractions defined by various kinds of patterns and parsers, bringing improvements to pattern matching, text processing, web automation, repetitive text editing, inference of patterns from examples, and error detection.

This dissertation makes contributions in a number of areas. Some contributions are theoretical:

- a model of text structure as region sets, which allow a variety of structure description mechanisms (regular expressions, grammars, or any kind of parser) to be encapsulated as simple abstractions;
- an algebra for region sets that enables structure abstractions to be composed and reused;
- data structures and algorithms for efficient representation of region sets and implementation of the region algebra;
- theoretical results about the classes of languages recognized by the region algebra;
- algorithms for inferring patterns from examples using lightweight structure;
- heuristics for repetitive editing that produce accurate inferences with few examples;
- algorithms for outlier finding that detect inconsistent pattern matches using lightweight structure.

Other contributions fall into the category of new languages and system designs:

- the TC pattern language, based on the region algebra, which allows users to write simple, readable text patterns using structure abstractions;
- a command language that extends Unix-style text processing to richly-structured text like web pages and source code;
- the browser shell, which integrates a command prompt into the web browsing interaction model.

Finally, several contributions are made to user interface design:

- various techniques for making multiple selections in text, including mouse selection, pattern matching, and combinations thereof;
- using multiple selections for interactive text editing;
- using mouse selection to give positive and negative examples of text patterns;
- using multiple selections and patterns to give feedback about inference;
- highlighting techniques for displaying region sets in a document;
- highlighting techniques that draw the user's attention to unusual selections or pattern matches, in order to reduce errors.

1.5 Thesis Overview

The next chapter, Chapter 2, surveys related work. Then the heart of the dissertation is divided into two parts. The first part describes lightweight structure itself:

- Chapter 3 defines the region set model and the region algebra, and shows how higher-level pattern-matching operators can be defined in terms of the core algebra.
- Chapter 4 shows how the region set model and algebra can be implemented efficiently.
- Chapter 5 proves some results about the expressive power of the region algebra, characterizing the classes of languages recognized by region algebra expressions that use various kinds of structure abstractions.

The second part discusses applications of lightweight structure:

- Chapter 6 describes text constraints (TC), the user-level pattern language based on the region algebra.
- Chapter 7 introduces the LAPIS web browser/text editor. A key feature of LAPIS is its support for multiple selections, represented by a region set. This chapter describes how multiple selections are made in LAPIS using the mouse, the pattern library, and TC patterns. A user study tested all three selection mechanisms, focusing in particular on the readability and writability of TC patterns.
- Chapter 8 describes the text-processing commands built into LAPIS. LAPIS includes commands that are similar to familiar Unix text-processing tools like `grep` and `sort`, but are more useful on richly-structured text like HTML and source code. This chapter also describes the browser shell, which integrates a command interpreter into the web browsing model, and shows how LAPIS can be used to automate interactions with web sites.

- Chapter 9 explains how LAPIS infers multiple selections from examples using two different techniques: selection guessing and simultaneous editing. Two user studies included in this chapter show that selection guessing and simultaneous editing help even on small repetitive tasks, and are more effective than another repetitive-text-editing system, DEED [Fuj98].

- Chapter 10 explains how outlier finding is used in LAPIS to highlight possible errors in a selection. A user study showed that outlier highlighting reduced the tendency of users to overlook inference errors.

Finally, Chapter 11 discusses some of the major design decisions in LAPIS, reviews the contributions of the dissertation, and outlines future directions for this work.

Chapter 2

Related Work

This chapter surveys previous work in structured text processing. A common theme in much of this work is a choice between two approaches: the *syntactic* approach and the *lexical* approach.

In general terms, the syntactic approach uses a formal, hierarchical definition of text structure, invariably some form of grammar. Syntactic systems generally parse the text into a tree of elements, called a *syntax tree*, and then search, edit, or otherwise manipulate the tree.

The lexical approach, on the other hand, is less formal and rarely hierarchical. Lexical systems generally use regular expressions or a similar pattern language to describe structure. Instead of parsing text into a hierarchical tree, lexical systems treat the text as a sequence of flat segments, such as characters, tokens, or lines.

The syntactic approach is generally more expressive, since grammars can capture aspects of hierarchical text structure, particularly arbitrary nesting, that the lexical approach cannot. However, the lexical approach is generally better at handling structure that is only partially described. Other differences between the two approaches will be seen in the sections below.

2.1 Parser Generators

Parser generators are perhaps the earliest systems for describing and processing structured text. Lex [Les75] generates a lexical analyzer from a declarative specification consisting of a set of regular expressions describing tokens. Each token is associated with an action, which is a statement or block of statements written in C. A Lex specification is translated into a finite state machine that reads a stream of text, splits it into tokens, and invokes the corresponding action for each token. Lex is a canonical example of the lexical approach to text processing.

Lex is often used as a front-end for Yacc [Joh75]. A Yacc specification consists of a set of context-free grammar productions, each of which is associated with an action written in C. Yacc translates this specification into a parser that executes the associated action whenever it parses a production. In a Yacc specification designed for a compiler, the actions typically generate tree nodes and connect them together into a tree. Yacc is a good example of the syntactic approach.

Since Lex and Yacc are frequently used in concert — Lex for tokenization and Yacc for parsing — recent systems have recognized the value of combining both facilities into a single specification. In the parser generators JavaCC [Jav00] and ANTLR [Sch99], a single specification describes both tokens and nonterminals, the former by regular expressions and the latter by grammar productions.

JavaCC and ANTLR also support annotations that generate tree nodes automatically, so that there is no need to write action code for the common case of generating a syntax tree.

Parser generators like Lex and Yacc do not encourage reuse, because the structure descriptions — regular expressions and grammar productions — are mixed up with the actions that implement a particular task. A Yacc grammar designed for one task must be substantially modified to be applied to another task, even if the text structure is the same in both cases. As a result, parser generators are generally used for building compilers and tools that merit the substantial investment in developing or adapting the specifications. Parser generators are less often used for one-off tasks like searching source code, calculating code metrics, and global search-and-replace. In this sense, parser generators are a *heavyweight* approach to describing and processing text structure.

Lightweight structure offers a way to reuse the structure knowledge implicit in a parser generator specification. After a one-time investment of resources to retarget the specification so that it produces region sets, the generated parser can be installed in the structure library and used in a variety of tasks with no further development effort. Java syntax was added to LAPIS in precisely this way, using a Java parser generated by JavaCC (Section 6.2.4).

2.2 Markup Languages

Markup languages, such as SGML [Gol90] and its successor XML [W3C00], are a popular way to represent structured documents. SGML/XML use explicit delimiters, called *tags*, to represent structure in the document. A specification called a *document type definition* (DTD) defines the markup tags and specifies legal combinations of tags. A DTD is essentially a grammar over tags; the structure of text between tags is not described by the DTD. Given a document and a DTD that describes it, SGML/XML tools can parse the document, *validate* it (checking its syntax), and manipulate the document in a variety of ways.

SGML has been used mainly for publishing and documentation. Its most widespread application is the HTML format found on the World Wide Web. HTML documents are SGML documents that conform to the HTML DTD (although, strictly speaking, many HTML web pages do not actually conform). XML, which is a simpler subset of SGML, is seeing much wider application — not only for delivering hypertext content, but also for configuration files, application data files, remote procedure calls, and online data exchange.

Markup languages separate structure description, represented by the DTD, from structure manipulation, represented by the tools that operate on structured documents. Lightweight structure provides a similar separation. However, markup languages require explicit markup in the text to be processed. Text which is not marked up, like source code or plain text, cannot be processed.

One solution to this problem is to add explicit markup automatically. This approach is taken by JavaML [Bad99], which translates Java source code into XML. JavaML has been shown to be useful for tasks like calculating source code metrics, renaming variables and methods safely, and adding instrumentation code. However, JavaML requires its user to buy into a new representation for Java source code which is not shared by other Java tools. The user could convert back and forth between JavaML and conventional Java syntax, but the conversion process only preserves syntactic features, not lexical features like whitespace, indentation, and comment placement.

Even when explicit markup is available in a document, it may not completely describe the text structure. For example, suppose a DTD specifies that phone numbers are identified by a `<phonenumber>` tag, like this:

```
<phonenumber>+1 412-247-3940</phonenumber>
```

This level of markup allows entire phone numbers to be located and manipulated, but it does not represent the internal structure of a phone number, such as the country code or area code. Markup languages must make a tradeoff between ease of authoring and ease of processing — minimal markup makes authoring easier, but more thorough markup makes more structure available for processing.

Lightweight structure can use explicit markup to define structure abstractions, without sacrificing the ability to describe structure in other ways, such as regular expressions. LAPIS includes an HTML parser in its library.

2.3 Structured Text Editors

A number of tools have been designed to tackle the problem of editing structured text, particularly program source code. The evolution of structured text editors highlights some of the key differences between the syntactic and lexical approach.

Early structured editors, often called *syntax-directed editors*, followed the syntactic approach. Examples include Emily [Han71], the Cornell Program Synthesizer [TRH81], and Gandalf [HN86]. In a syntax-directed editor, text structure is specified by an annotated grammar that describes both the *parsing* process, which converts text into an abstract syntax tree, and the *unparsing* process, which renders the abstract syntax tree into formatted text for display on the screen. In a syntax-directed editor, the user can only select and edit complete syntactic units that correspond to subtrees, such as expressions, statements, or declarations. New tree nodes are created by commands that produce a template for the user to fill in. In order to make structural changes, e.g., changing a `while` loop into a `repeat-until` loop, the user must either apply an editing command specialized for the transformation, or else recreate the loop from scratch. The awkwardness of this style of editing is one of the criticisms that has been leveled at syntax-directed editors.

Other editors take a lexical approach to editing program source code. Rather than representing language syntax in detail, editors like Z [Woo81] and EMACS [Sta81] use lexical descriptions to provide some of the features of a syntax-directed editor while allowing the user to edit the program text in a freeform way. For example, when the user presses Enter to start a new line in a C program, Z automatically indents the next line depending on the last token of the previous line. For example, if the last token is `{`, then the next line is indented one level deeper than the previous line. Similarly, EMACS uses regular expressions to render program syntax in different fonts and colors, and offers commands that select and step through program syntax like statements and expressions. All these editing features are based only on local, lexical descriptions of structure, not on a global parse of the program, so these editors can be fooled into highlighting or selecting syntactically incorrect structure.

Defenses of each approach have been offered by Waters [Wat82] and Minor [Min92]. Arguing for plain text editors at a time when syntax-directed editors were in vogue, Waters addressed some commonly-voiced objections to plain text editing:

- *The text-oriented approach is obsolete.* Waters answers this objection by pointing out that plain text editing is familiar to users, and there is no point in throwing away an effective technique without good reason. His argument resounds even more strongly today, since an enormous base of computer users have become intimately familiar with plain text editing from email, instant messaging, and the World Wide Web. It seems preferable to use techniques that most users already know and understand.

- *Text-oriented editing is dangerous, because it does not guarantee syntactic consistency.* Waters points out that a plain text editor can use a parser to check syntax and highlight syntax errors in context, so syntax errors can at least be corrected. Unlike the syntax-directed approach, however, the text-oriented approach cannot prevent the errors in the first place.

- *Text-oriented editing is incompatible with syntax-directed editing.* In other words, implementing both approaches in the same editor is too hard. Waters agrees that hybrid editors are difficult but not impossible to build. In fact, several hybrid editors have been built since then, some of which are discussed below.

Arguing for the other side a decade later, after most programmers had chosen text-oriented editors over syntax-directed ones, Minor took up some common objections to syntax-directed editing:

- *Syntax-directed editing is awkward, particularly for expressions.* Minor responds to this point by arguing for a hybrid approach in which expressions are edited as plain text and then reparsed into a syntax tree. This hybrid approach is described in more detail below.

- *Enforced syntactic consistency limits the user's freedom to pass through inconsistent intermediate states.* Minor answers this objection by claiming that users do not care about this freedom in and of itself; what they care about is convenient editing. If syntax-directed operations can be made as convenient as plain text editing, then users will not care whether the editor permits syntactic inconsistency.

- *Expert users don't need syntactic guidance.* Syntax-directed editing is particularly helpful for novice programmers because it prevents syntax errors, by design. For experts, however, this assistance may be less valuable, and possibly inhibiting. Minor suggests the tradeoff here is similar to the tradeoff between menus and command languages in user interface design. Menus are easier for novices to learn, but commands allow experts to achieve higher performance. Well-designed systems should support both novices and experts, so Minor argues for a hybrid editor that provides syntax-directed editing for novices but less constrained editing for experts.

In fact, both Waters and Minor were arguing for the same thing: a hybrid approach that combines the best features of syntax orientation and text orientation. Several editors have taken this approach, either by adding freeform text editing to a primarily syntax-directed editor, or vice versa. The Synthesizer Generator [RT89], the successor of the Cornell Program Synthesizer, is a syntax-directed editor that permits limited freeform text editing. This editing is done by selecting a subtree of the syntax tree — such as an expression, a statement, or a function body — and editing its unparsed representation. When the user signals that editing is complete, usually by navigating to another subtree in the program, the edited representation is reparsed into tree nodes. Thus the Synthesizer Generator's primary representation is a tree. In contrast, Pan [VGB92] is a text-oriented editor, with an incremental parser that creates a syntax tree on demand when the user invokes syntax-directed commands.

Lightweight structure offers another kind of hybrid approach. Like the text-oriented approach, the primary representation in lightweight structure is a string of characters, not a tree. Syntax-directed features are available on demand, implemented by parsers or patterns in the structure library. Unlike other hybrids, however, lightweight structure also combines the structure description methods of the two approaches. Structure can be described in syntactic fashion, as in syntax-directed editors, or in lexical fashion, as in text-oriented editors, or in some combination, according to the level of convenience, tolerance, or precision demanded by a particular task.

LAPIS is not a complete programming editor, like EMACS or Gandalf. LAPIS does not provide all the features programmers have come to expect from such editors, such as automatic indentation, syntax highlighting, and template generation. Although these features could be provided by lightweight structure, they lie outside the application areas targeted by this thesis, so they are left for future work.

2.4 Structure-Aware User Interfaces

A recent trend in structured text processing are systems that use recognized structure to trigger user interface actions. This category, which might be called *structure-aware user interfaces*, includes the Intel Selection Recognition Agent [PK97], Cyberdesk [DAW98], Apple Data Detectors [NMW98], LiveDoc [MB98a], and Microsoft Smart Tags [Mic02].

A structure-aware user interface has a collection of *structure detectors* (e.g., for email addresses or URLs), plus a collection of *actions* that can be performed on each kind of structure (e.g., sending an email message or browsing to a URL). When the user makes a text selection in an application, the system automatically runs structure detectors on the selection and builds a list of possible actions. The list of actions is then popped up as a menu.

The structure detectors in a structure-aware user interface can be implemented by arbitrary patterns and parsers, just like lightweight structure abstractions. Unlike lightweight structure, however, these structure detectors are not designed for reuse. For example, the structure detector for email addresses cannot be used in a search-and-replace pattern, or constrained so that it only matches email addresses from `cmu.edu`.

2.5 Pattern Matching

The oldest text pattern language is probably SNOBOL [GPP71], a string manipulation language with an embedded pattern language. Its successor was Icon [GG83], which integrated pattern matching more tightly with other language features by supporting goal-directed evaluation. Icon allows programmers to define their own pattern-matching operators, a degree of expressiveness seen in few other pattern-matching systems.

Today, most text-processing languages use regular expressions for text pattern matching, the most prominent examples being awk [AKW88] and Perl [WCS96]. Regular expressions are fast, easy to implement, and extremely compact, at the cost of being cryptic. In an effort to address this problem, several researchers have designed visual languages for regular expressions [JB97, Bla01], finding that users can generate and comprehend the visual representations better than the standard textual representations. This research does not address more fundamental problems with regular expressions, however — namely, that regular expressions are too low-level, insufficiently expressive, and fail to encourage reuse.

The preceding pattern matching systems take a lexical approach, searching for patterns in a string of characters. *Structured text query languages* take a syntactic approach instead, searching for patterns in a syntax tree. A variety of query languages of this sort have been proposed, including p-strings [GT87], Maestro [Mac91], PAT expressions [ST92], tree inclusion [KM93], Proximal Nodes [NBY95], GC-lists [CCB95], and sgrep [JK96]. Baeza-Yates and Navarro give a good survey of structured text query languages [BYN96]. Some structured text query languages are designed expressly for searching SGML/XML markup, such as OmniMark [Omn99].

Markup systems have found other uses for pattern matching beyond searching. For example, DSSSL [Cov96], CSS [BLLJ98], and XSLT [Kay01] use patterns in stylesheets to describe how SGML, HTML, and XML documents (respectively) should be rendered on screen or on paper. Each rule in a stylesheet consists of a pattern and a set of rendering properties, such as color, font, or positioning information. Elements that match the pattern are assigned the corresponding properties. The pattern can constrain not only the element type, but also its attributes, its context (parent element), and its content (child elements and text). In XML, patterns are also used for hyperlinking. XLink [DMO01] uses the same pattern language as XSLT to specify hyperlink targets in XML documents.

Several pattern languages have been proposed for searching and transforming source code. Some take the syntactic approach, in which patterns are matched against a syntax tree produced from the source. Tawk [GAM96] uses the pattern-action model of awk, but patterns are syntactic tree patterns instead of regular expressions. ASTLOG [Cre97] is a Prolog-based pattern language in which pattern operators are Prolog predicates over tree nodes. Like Icon, ASTLOG allows users to define new pattern operators.

The lexical approach has also been used in source code processing, notably by the Lexical Source Model Extraction (LSME) system [MN96]. LSME is a pattern-action system similar to awk, in which patterns are regular expressions over program tokens, tested one line at a time, and actions consist of Icon code. Unlike awk, however, LSME supports hierarchical rules, in which child rules do not become active until the parent rule has matched. Hierarchical rules make it possible to generate a procedure call graph, for example, using a parent rule to match a procedure declaration and a child rule to find calls to other procedures within its body.

Finally, it is worth mentioning *aspect-oriented programming* [KLM$^+$97], which uses pattern matching to insert code at various points in an object-oriented program. Unlike the previous systems discussed in this section, the patterns used in aspect-oriented programming can incorporate semantic information as well as syntactic. Aspect-oriented patterns can also refer to dynamic properties of a running program, such as the ancestors of the current method on the call stack. Other source code transformation systems — including LAPIS — are entirely static.

In comparison to all these pattern languages, the text constraints (TC) language described in this thesis is the first text pattern language that combines the syntactic and lexical approaches into a single unified framework. TC includes some operators that are syntactic in nature — particularly the relations *in* and *contains*, which appear in most structured text query languages, but not in regular expressions. It also includes operators that are lexical in nature — particularly concatenation, which is a fundamental operation in regular expressions but is impossible to represent with a tree-based pattern that can only match one tree node. Furthermore, the region algebra underlying TC can express many of the operators found in these other pattern languages, although not all have been implemented in TC. More details about these operators can be found in Chapter 3.

However, the region algebra cannot express arbitrary repetition, like the Kleene star used in regular expressions. As a result, TC expressions that use only literal string matching are weaker than regular expressions (Chapter 5). In practice, TC expressions are used to combine and reuse structure abstractions defined by more powerful pattern languages, such as regular expressions and grammars, so this weakness is not a serious problem. Nevertheless, adding the Kleene star to the region algebra would be an interesting avenue of future work.

2.6 Web Automation

Web automation — automatically browsing the World Wide Web and extracting data – is one of the problems to which lightweight structure is applied by this thesis. Another approach to this problem is *macro recording*, typified by LiveAgent [Kru97]. LiveAgent automates a web-browsing task by recording a sequence of browsing actions in Netscape through a local HTTP proxy. Macro recording requires little learning on the part of the user, but recorded macros suffer from limited expressiveness, often lacking variables, conditionals, and iteration.

Another approach is *scripting*, writing a program in a scripting language such as Perl, Tcl, or Python. These scripting languages are fully expressive, Turing-complete programming languages, but programs written in these languages must be developed, tested, and invoked outside the web browser, making them difficult to incorporate into a web user's work flow. The overhead of switching to an external scripting language tends to discourage spur-of-the-moment automation, in which interactive operations might be mixed with automation in order to finish a task more quickly.

A particularly relevant scripting language is WebL [KM98], which provides high-level *service combinators* for invoking web services and a *markup algebra*, essentially a structured text query language, for extracting results from HTML. Like other scripting languages, WebL lacks tight integration with a web browser, forcing a user to study the HTML source of a web service to develop markup patterns and reverse-engineer form interfaces. In LAPIS, web automation can be performed while viewing rendered web pages in the browser, and simple tasks can be automated by demonstrating the steps on examples.

Other systems have tackled web automation by demonstration. Turquoise [MM97] and Internet Scrapbook [SK98] construct a *personalized newspaper*, a dynamic collage of pieces clipped from other web pages, by generalizing from a cut-and-paste demonstration. SPHINX [MB98b] creates a web crawler by demonstration, learning which URLs to follow from positive and negative examples.

2.7 Repetitive Text Editing

Users have a rich basket of tools for automating repetitive editing tasks. *Find-and-replace*, in which the user specifies a pattern to search for and replacement text to be substituted, is good enough for simple tasks. *Keyboard macros* are another technique, in which the user records a sequence of keystrokes (or editing commands) and binds the sequence to a single command for easy reexecution. Most keyboard macro systems also support simple loops using tail recursion, where the last step in the macro reinvokes the macro. For more complicated tasks, however, users may resort to writing a script in a text-processing language like awk or Perl.

Sam [Pik87] and its successor Acme [Pik94] combine an interactive editor with a command language that manipulates regions matching regular expressions. Regular expressions can be pipelined to automatically process multiline structure in ways that line-oriented systems like awk cannot. Unlike LAPIS, however, Sam and Acme do not provide mechanisms for naming, composing, and reusing the structure described by the regular expressions.

Another approach to the problem of repetitive text editing is *programming by example,* also called *programming by demonstration* (PBD). In PBD, the user demonstrates one or more examples of the transformation in a text editor, and the system generalizes this demonstration into a program that can be applied to the rest of the examples. PBD systems for text editing have included EBE [Nix85], Tourmaline [Mye93], TELS [WM93], Eager [Cyp93], Cima [Mau94], DEED [Fuj98], and SmartEDIT [LWDW01].

Unlike LAPIS, none of these systems used multiple selection for editing or feedback about inferences. Multiple selections completely reshape the dialog between a PBD system and its user. While a traditional PBD system reveals its predictions one example at a time, multiple selections allow the system to expose all its predictions simultaneously. The user can look and see that the system's inference is correct, at least for the current set of examples, which in many tasks is all that matters. Novel forms of intelligent assistance, such as outlier highlighting, can help the user find inference errors. The user can correct erroneous predictions in *any* order, not just the order chosen by the PBD system. Alternative hypotheses can be presented not only abstractly, as a data description or pattern, but also concretely, as a multiple selection. If the desired concept is unlearnable, the user may still be able to get close enough and fix the remaining mispredictions by hand, without stopping the demonstration.

The system that most closely resembles multiple-selection editing is Visual Awk [LH95]. Visual Awk allows a user to create awk-like file transformers interactively. Like awk, Visual Awk's basic structure consists of lines and words. When the user selects one or more words in a line, the system highlights the words at the same position in all other lines. For other kinds of selections, however, the user must select the appropriate tool: e.g., Cutter selects by character offset, and Matcher selects matches to a regular expression. In contrast, LAPIS is designed around a conventional text editor, operates on arbitrary records (not just lines), uses standard editing commands like copy and paste, and infers selections from examples.

2.8 Inferring Text Patterns From Examples

LAPIS can infer a text pattern from multiple examples given by the user (Chapter 9). Several other authors have tackled this problem as well. Kushmerick [KWD97] showed how to learn lexical patterns, which he called "wrappers", that extract data from structured text like web pages and seminar announcements. Freitag [Fre98] compared several approaches to learning wrappers, including a naive Bayesian classifier and a relational learner. Both Kushmerick and Freitag use statistical methods that need a large number of labeled examples to learn a concept, often 20 or more. This approach is hard to apply in an interactive interface, where user become frustrated when forced to give more than a handful of examples.

Most PBD systems for text editing infer patterns from examples. The most advanced PBD inference algorithm is probably Cima [Mau94], which combines a powerful disjunctive normal form (DNF) learner with a dynamic bias that allows Cima to vary the space of features and hypotheses it considers in response to *hints* given by the user. Another PBD approach is found in the Grammex system [LNW98], which infers context-free grammars from a small number of examples chosen and annotated by the user.

Compared to these systems, LAPIS is the first that can learn patterns over arbitrary structure abstractions. Other approaches learn from a limited set of low-level features, such as literals, character classes (e.g., alphanumeric or digit), and capitalization. Patterns learned by LAPIS can use any abstraction in the structure library, including complex syntax like Java and HTML that would be infeasible to learn from examples.

2.9 Error Detection

LAPIS includes a novel feature called *outlier finding* that uses lightweight structure to detect unusual selections that might be erroneous (Chapter 10). The notion of an outlier comes from the field of statistics, which defines it as a data point which appears to be inconsistent with the rest of the data. Most work on outliers focuses on statistical tests that justify omitting outliers from experimental data [BL84]. A large number of statistical tests have been developed for various probability distributions. For outliers in text selections or pattern matches, however, the distribution is rarely simple and almost always unknown. The LAPIS outlier finder cannot make strong statistical claims like “this outlier is 95% likely to be an error,” but on the other hand it can be applied more widely, with no assumptions about the distribution of the data.

Outlier finding has been applied to data mining by Knorr and Ng [KN98]. They propose a “distance-based” definition of an outlier, which is similar to the approach taken in this thesis. They define a $DB(p, D)$ outlier as a data object that lies at least a distance D (in feature space) from at least a fraction p of the rest of the data set. The choice of p and D is left to a human expert. The algorithm described in this thesis is simpler for inexpert users because it merely ranks outliers along a single dimension of “weirdness”. Users don’t need to understand the details of the outlier finder to use it, and appropriate weights and parameters are determined automatically by the algorithm.

The outlier finding algorithm draws on techniques better known in the machine learning community as *clustering* [And73]. In clustering, objects are classified into groups by a similarity measure computed from features of the object. Clustering is commonly used in information retrieval to find similar documents, representing each document by a vector of terms and computing similarity between term vectors by Euclidean distance or cosine measures. LAPIS is concerned with matching small parts of documents rather than whole documents, so term vectors are less suitable as a representation. Freitag confirmed this hypothesis in his study of inductive learning for information extraction [Fre98], which showed that a relational learner using features similar to the LAPIS outlier finder was much more effective at learning rules to extract fields from text than a term-vector-based learner.

One way to find borderline mismatches in text pattern matching is to allow errors in the pattern match. This is the approach taken by *agrep* [WM92], which allows a bounded number of errors (insertions, deletions, or substitutions) when it matches a pattern. Agrep is particularly useful for searching documents which may contain spelling errors.

Spelling and grammar checking are well-known ways to find errors in text editing. Microsoft Word pioneered the idea of using these checkers in the background, highlighting possible errors with a jagged underline as the user types. Although spell-checking and outlier-finding both have the same goal – reducing errors – the approaches are drastically different. Spelling and grammar checkers compare the text with a known model, such as a dictionary or language grammar. Outlier finding has no model. Instead, it assumes that the text is mostly correct already, and searches for exceptions and irregularities. Whereas a conventional spell-checker would be flummoxed by text that diverges drastically from the model — such as Lewis Carroll's "The Jabberwocky" [Car72] — a spell checker based on outlier-finding might notice that one occurrence of "Jabberwock" has been mistyped because it is spelled differently from the rest. On the other hand, an outlier-finding spell checker would overlook systematic spelling errors. Morris and Cherry built an outlier-finding spell-checker [MC74] that computes trigram frequencies for a document and then sorts the document's words by their trigram probability, and found that it worked well on technical documents. The LAPIS outlier finder has not been applied to spell-checking, but doing so would be interesting future work.

Chapter 3

Region Algebra

The fundamental unit of lightweight structure is a contiguous segment of text, called a *region*. Other names for the same concept are runs, substrings, and pieces [KM98]. Many features of text-processing systems operate on regions:

- **Styles.** In a word processor, styles like font, color, boldface, italics, underlining, or strikeout are associated with regions.
- **Selection.** When a user selects text with a mouse, the selection is a region. The text-insertion point may also be represented as a *zero-length* region, i.e., a region whose start point and end point are identical.
- **Hyperlinking.** In a web browser, hyperlinks are represented as clickable text regions. The target of a hyperlink is also a region, either in the same web page or a different page.
- **Pattern matching.** The result of searching for a pattern in a text editor or browser is a region that matches the pattern.
- **Logical document structure.** Chapters, sections, paragraphs, sentences, and words can all be represented by regions.
- **Physical document structure.** Pages, frames, columns, and lines can be represented by regions.
- **Language syntax.** Expressions and statements in source code and phrases in natural language can be represented by regions.

The lightweight structure model is mainly concerned with *sets* of regions, because a structure abstraction returns a set of regions. Figure 3.1 shows some examples of region sets.

The goal of this chapter is to develop an algebra for combining sets of regions. The algebra will enable a number of implementation alternatives and optimizations (Chapter 4), and will serve as the basis of a user-level pattern language (Chapter 6). The region algebra plays the same role in lightweight structure that relational algebra does in relational databases: relational algebra is a formal model of relational data that acts as a target for implementations (database engines) and the basis of a user-level language (SQL).

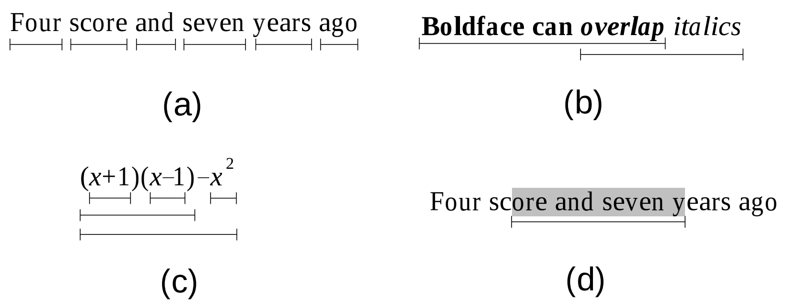

Four score and seven years ago
Boldface can overlap italics
(a)
(b)
(x+1)(x–1)–x2
Four score and seven years ago
(c)
(d)

Figure 3.1: Example region sets. (a) words; (b) styles; (c) expressions; (d) a selection made by a user.

The applications listed above suggest some properties that need to be captured by the region algebra:

- **Regions may be nested in other regions**. In documents, words are completely contained in a sentence, sentences in a paragraph, and paragraphs in a section. In programs, expressions are nested in other expressions, as in Figure 3.1(c). Natural language also obeys a nested structure: noun phrases are nested in verb phrases, and verb phrases in clauses or sentences.

- **Regions may overlap other regions.** Logical structure can overlap physical structure in complex ways. For example, sentences and lines often overlap without nesting. Styles may also overlap, as in Figure 3.1(b).

- **Regions may be zero-length.** In a text editor, it is often convenient to treat every user selection as a region, including the text insertion caret that appears *between* two characters. Some structuring attributes may also be represented as zero-length regions, such as hyperlink targets, page breaks, and hyphenation hints.

Since text structure is so varied — nested, overlapping, zero-length — the region algebra should be capable of representing *arbitrary* sets of regions. Unfortunately, the size of an arbitrary region set may be quadratic in the length of the string. For a simple example, consider the set of all regions in a string of length n. There are n regions starting at the beginning of the string, $n-1$ regions starting after the first character, etc. Thus the total number of regions is $\sum_{i=1}^{n} i = n(n+1)/2$, which is $O(n^2)$. Quadratic region sets cause problems for efficient implementation. Previous systems have dealt with this problem by restricting region sets to certain types that are guaranteed to be linear, sacrificing expressiveness for linear processing.

This chapter[1] takes a more general approach, describing an algebra that can combine arbitrary sets of regions. The general algebra has two main advantages. First, it is very simple, consisting only of a handful of relational operators, the standard set operators, and an iteration operator. The simplicity will make it easier to prove its theoretical power in Chapter 5. Previous systems needed far more operators. Second, supporting arbitrary region sets means that the algebra can combine structure found by arbitrary parsers. Users of the algebra need not be concerned with whether two region sets satisfy a particular restriction before combining them. This property helps human users

[1]Portions of this chapter and the next are adapted from an earlier paper [MM99].

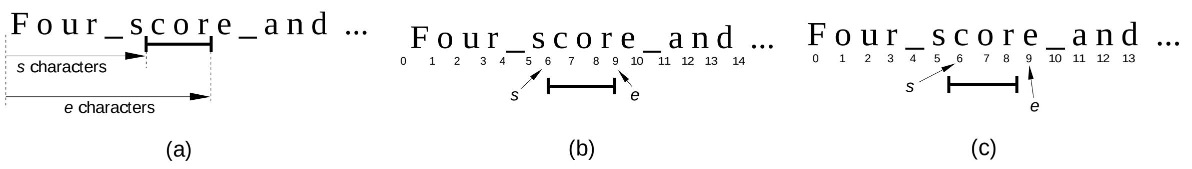


Figure 3.2: Equivalent definitions of a region $[s, e]$: (a) s and e are offsets from the start of the string; (b) s and e denote positions between characters, and $[s, e]$ is a closed interval; (c) s and e denote characters, and $[s, e]$ is a half-open interval.

by making it easier to write patterns (Chapter 6). It also helps machine learning agents by making it easier to define features and hypotheses (Chapters 9 and 10).

The next chapter will consider the question of implementation, showing that the general algebra can be implemented efficiently.

3.1 Regions

Given a string of length n, a *region* in the string is denoted by a pair $[s, e]$, where $0 \leq s \leq e \leq n$. The start offset s is the number of characters preceding the region's start point, and the end offset *e* is the number of characters preceding the region's end point (Figure 3.2(a)). The length of the region is $e - s$. If $s = e$, then the region has zero length, corresponding to a position between characters instead of a span of characters.

One can interpret a region $[s, e]$ in other ways that are equivalent to the definition above. If the positions between each character in the string are numbered 0, 1, 2, etc., then $[s, e]$ corresponds to a closed interval from position s to position e (Figure 3.2(b)). If the characters of the string are numbered instead, then $[s, e]$ corresponds to the half-open interval from character s to character e (Figure 3.2(c)). It should be clear that these definitions are all equivalent.

Another common way to describe a one-dimensional interval uses its start offset and *length,* instead of its end offset. Still another method, often seen in computational geometry, describes an interval by its *centroid* (the average of its start and end offsets) and length [HN83]. I prefer the definition given above, because it is easier to define relations like *before* and *after* if regions are described by end points rather than lengths.

3.2 Region Relations

The algebra is based on three fundamental binary relations among regions:

- $[s, e]$ *before* $[s', e']$ if and only if $e \leq s'$ and $s < s'$

- $[s, e]$ *overlaps-start* $[s', e']$ if and only if $s \leq s'$ and $e \leq e'$

- $[s, e]$ *contains* $[s', e']$ if and only if $s \leq s'$ and $e' \leq e$

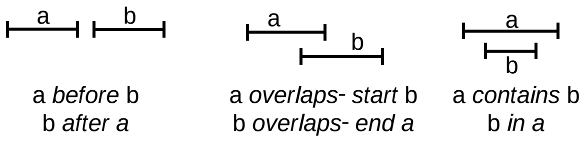


Figure 3.3: Fundamental region relations.

The *overlaps-start* and *contains* relations are reflexive, but *before* is not.[2] The *before* and *contains* relations are transitive, but *overlaps-start* is not. None of the relations are symmetric, so let us name their inverses as well:

- $[s, e]$ *after* $[s', e']$ if and only if $[s', e']$ *before* $[s, e]$
- $[s, e]$ *in* $[s', e']$ if and only if $[s', e']$ *contains* $[s, e]$
- $[s, e]$ *overlaps-end* $[s', e']$ if and only if $[s', e']$ *overlaps-start* $[s, e]$

All six relations are illustrated in Figure 3.3. Taken together, the six relations are complete in the following sense:

Claim 1. *For any two regions* $[s, e]$ *and* $[s', e']$*, there exists at least one of the six region relations,* op*, such that* $[s, e]$ *op* $[s', e']$.

The truth of this claim will be more obvious with the help of a two-dimensional, geometric interpretation of regions, described in the next section.

These relations between regions in a string are similar to the relations between time intervals defined by Allen [All83], which is natural because both domains involve intervals in one dimension. Allen's temporal relations use strict comparisons, however, so his *before* is defined as $[s, e]$ *before* $[s', e']$ if and only if $e < s'$. As a result, Allen must also include relations for equality and adjacency (e.g. $[s, e]$ *meets* $[s', e']$ if and only if $e = s'$), for a total of thirteen relations. In contrast, my formulation allows adjacency and equality to be expressed as Boolean combinations of the six fundamental relations, as will be seen below in Section 3.6.1. The choice between the two formulations is largely a matter of taste, however, since all of Allen's relations can be derived from my region relations and vice versa.

3.3 Region Space

Region space is a two-dimensional plane in which the x-coordinate is the start of a region and the y-coordinate is the end (Figure 3.4). A region $[s, e]$ corresponds to the point (s, e) in region space. Strictly speaking, region space does not occupy the entire real plane. Only points with integral coordinates correspond to regions, and only if they lie in the closed triangular area above the 45° line, where $0 \leq s \leq e \leq n$.

When the string is short, it is often convenient to write the characters of the string along the x and y axis, as is done in Figure 3.4. Since regions begin and end between characters, the characters

[2] In fact, *before* is defined so that no region can be *before* itself. This is the reason for the second inequality in the definition, $s < s'$, which prevents a zero-length region $[x, x]$ from being *before* itself.

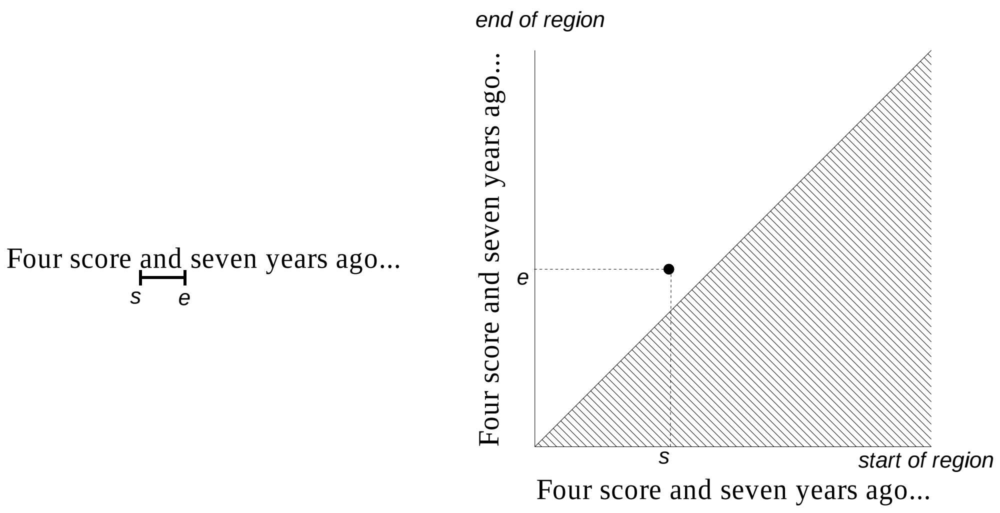

end of region
Four score and seven years ago...
Four score and seven years ago...
s
e
e
s
start of region
Four score and seven years ago...

Figure 3.4: Region space. A region $[s, e]$ corresponds to the point (s, e) in region space. Only integer-valued points on or above the $45°$ line are valid regions.

written along the axis actually label the intervals between region endpoints, rather than the region endpoints themselves.

The upper-left corner of region space corresponds to the region $[0, n]$, which spans the entire string. Points along the $45°$ diagonal represent zero-length regions in the string. The most interesting ones are the lower-left corner $[0, 0]$, which corresponds to the start of the string, and the upper right corner $[n, n]$, which is the end. The length of a region, $e - s$, is given by its y-distance above the $45°$ diagonal. Thus zero-length regions must lie on the diagonal itself, whereas the longest possible region, the entire string itself, must be the upper-left corner.

The region relations correspond to geometric relationships in region space. Suppose we hold some region $b = [b_s, b_e]$ fixed, and consider the set of all regions $a = [a_s, a_e]$ such that a *before* b. From the definition of *before,* it must be true that $a_e \leq b_s$. The set of regions a satisfying this constraint corresponds to the closed triangular area in region space shown in Figure 3.5(a). Parts (b) and (c) of the same figure show the corresponding closed rectangular areas for a *overlaps-start* b and a *contains* b, respectively.

Extending this analysis to include the three inverse relations *after, overlaps-end,* and *in* produces the region space "map" shown in Figure 3.6. The areas in this map are all closed. Each area includes its boundary, and adjacent areas intersect on their common boundaries.

The map completely covers region space, the triangular area above the diagonal line. Thus, for any region a, a must lie in some relation to b (or more than one, if a lies on a boundary between map areas). A similar map can be drawn for any choice of b. Points lying on the boundary of region space have degenerate maps, however (Figure 3.7). For example, Figure 3.7(a) shows the map for a zero-length region b lying on the diagonal. The areas for a *overlaps-start* b and *a overlaps-end b* have shrunk to lines, and the area for *a in b* has shrunk to the point b itself. The remaining parts of Figure 3.7 show the maps for other degenerate points in region space. In every case, the degenerate map still completely covers region space. Thus, a geometric argument suffices to establish Claim 1. For any pair of regions a, b, it must be the case that a lies in some area op in b's region space map because the map covers region space, so *a op b.*

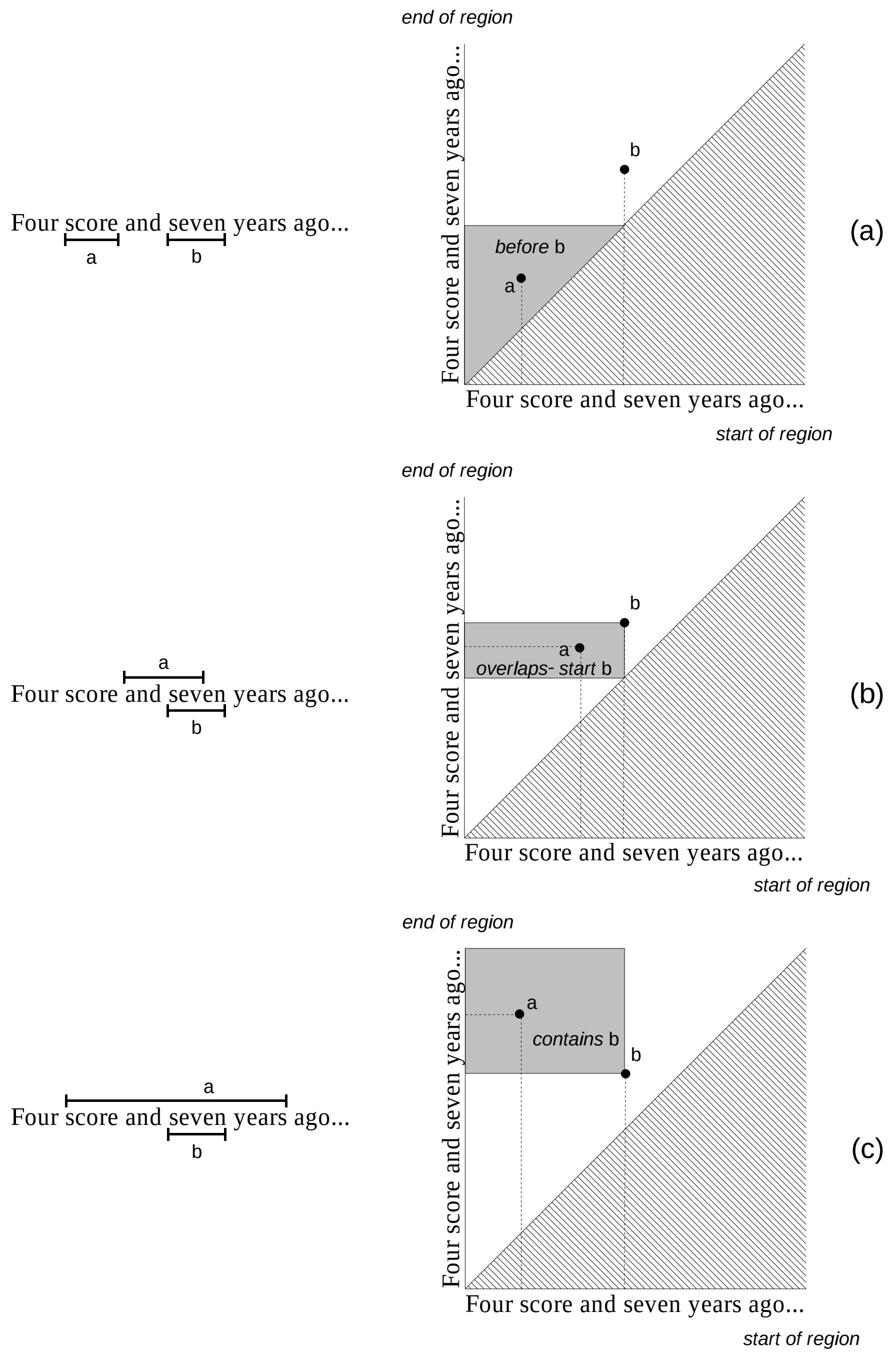


Figure 3.5: Areas of region space representing region relations relative to a fixed region b. The shaded rectangles represent all regions a satisfying a *before* b, a *overlaps-start* b, and a *contains* b, respectively.

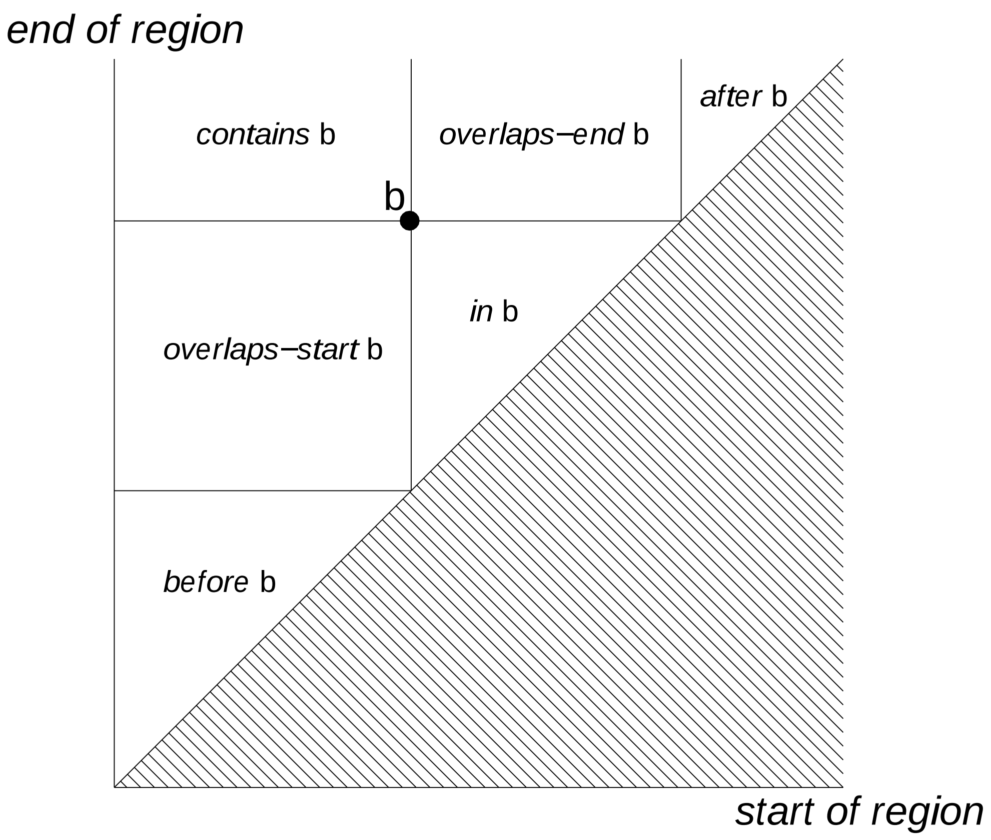


Figure 3.6: Region space map. "*op* b" labels the set of regions a such that a *op* b.

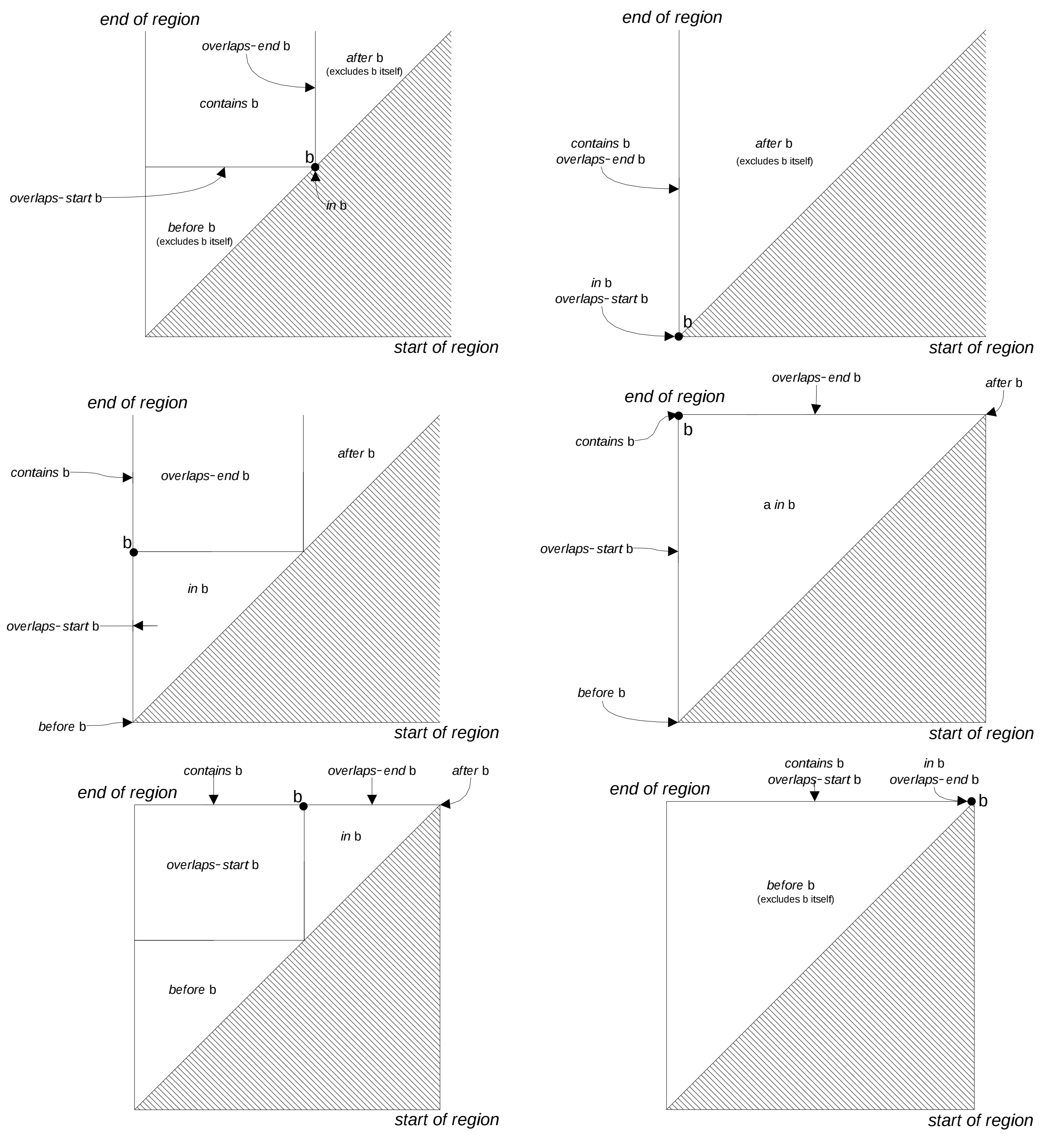


Figure 3.7: Degenerate region space maps, for regions b that lie on boundaries of region space.

Region space will be used repeatedly to illustrate the discussion of region algebra in this chapter and the next. Although it may take some effort to shift one's perspective from intervals along a one-dimensional string to points and areas in a two-dimensional plane, the rewards are worth it. A pictorial representation that shows every region as a clearly defined point makes it much easier to find boundary cases and ensure that a definition or algorithm handles them properly, as we did in Figure 3.7. Region space will also motivate the implementation of region algebra presented in the next chapter.

Transforming intervals in one dimension into points in two dimensions is not a new idea; the technique is well known in computational geometry [PS85] and spatial data structures [Sam90]. Rit [Rit86] presented a two-dimensional map of Allen's time interval relations that was very similar to the region space map shown in Figure 3.6, and Kulpa [Kul97] explored the maps that result when other interval representations are used, such as (centroid, length). However, this work is apparently the first that applies the transformation to text processing.

3.4 Region Set Types

The region relations can be used to define some important classes of region sets. If A denotes a set of regions, then:

- A is **flat** if and only if for every $a, b \in A$, at least one of the following holds: a *before* b; a *after* b; or $a = b$.
- A is **nested** if and only if for every $a, b \in A$, at least one of the following holds: a *before* b; a *after* b; a *in* b; a *contains* b; or $a = b$.
- A is **overlapping** if and only if for every $a, b \in A$, at least one of the following holds: a *before* b; a *after* b; a *overlaps-start* b; a *overlaps-end* b; or $a = b$.

From the definitions, it should be clear that A is flat if and only if A is both nested and overlapping. It also follows directly that if A is flat, nested, or overlapping, then all subsets of A are flat, nested, or overlapping, respectively.

An example of each kind of region set can be found in Figure 3.1. The words in Figure 3.1(a) are a flat set, the styles in Figure 3.1(b) are an overlapping set, and the expressions in Figure 3.1(c) are a nested set. Flat sets and nested sets are generally more common in text structure than overlapping sets.

Flat, nested, and overlapping region sets are particularly important because they are always linear in the length of the string, as the following theorems show.

Theorem 1. *If A is an overlapping region set in a string of length n, then $|A| \leq 2n + 1$.*

Proof. Consider A as a set of points in region space. Sweep a -45° line ($s + e = k$ for all integers k) across region space, from the lower left corner to the upper right corner. See Figure 3.8. When the sweep line intersects some point $a \in A$, then one part of the sweep line lies in *contains* a and the other part in *in* a, but the region space areas for *before* a, *after* a, *overlaps-start* a, and *overlaps-end* a touch the line only at a, so the sweep line can intersect no other points in A. Therefore every sweep position of the line intersects at most one region in A. Since there are $2n + 1$ sweep positions (k can range from 0 to $2n$), there are at most $2n + 1$ regions in A.

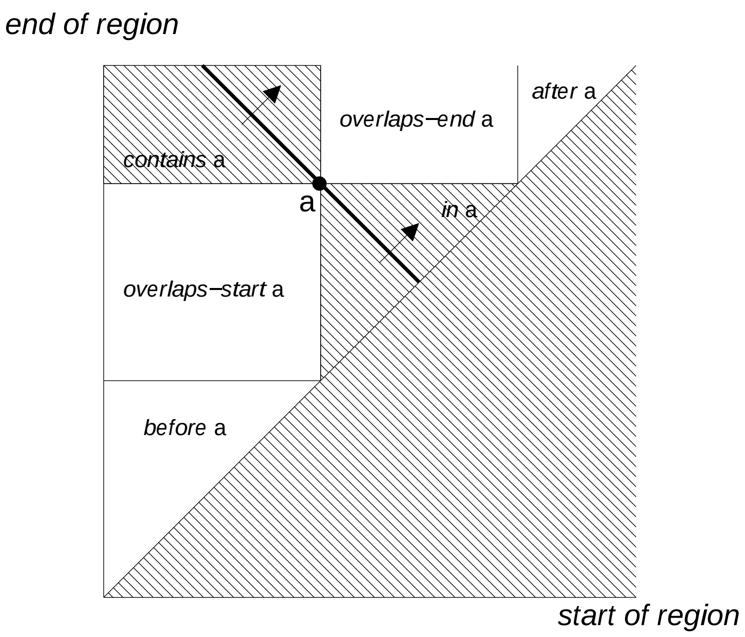


Figure 3.8: When a $-45°$ line is swept across an overlapping region set, it can intersect at most one region at every sweep position, because the rest of the line lies in the off-limits *in* and *contains* areas.

□

Corollary 1. *If A is a flat set, then $|A| \leq 2n + 1$.*

This bound is tight, because the set of all length-0 regions $[k, k]$ and length-1 regions $[k, k+1]$ is a flat set of size $2n + 1$.

Theorem 2. *If A is a nested region set in a string of length $n > 0$, then $|A| \leq 3n$.*

Proof. The following argument shows, by induction on n, that if A is a nested set with no zero-length regions, then $|A| \leq 2n - 1$. Since there are at most $n + 1$ zero-length regions in a string of length n, it will follow that an arbitrary nested region set can have at most $3n$ elements.

When $n = 1$, the only possible nonzero-length region is $[0, 1]$, so $|A| \leq 1$ as desired. Now suppose that all nested sets on strings of length $n' < n$ have at most $2n' - 1$ nonzero-length regions, and consider a nested set A with no zero-length regions on a string of length $n > 1$. There are two cases:

1. $[0, n] \notin A$. The following argument shows that there exists some k, $0 < k < n$, that partitions A such that every region in A is either *before* or *after* $[k, k]$. If A is empty, then this is trivially true. Otherwise, let r be the longest region in A (choosing arbitrarily if there's a tie). Some endpoint of r must lie strictly between 0 and n (otherwise $r = [0, n]$, which was previously excluded). Let k be this endpoint, and assume without loss of generality that k is the start of r. Because A is a nested set, for any region $a \in A$, at least one of the following holds:

   (a) a *before* r, which implies that a *before* $[k, k]$;

   (b) a ***after*** r, which implies that a ***after*** $[k, k]$;

   (c) a ***in*** r, which implies that a ***after*** $[k, k]$;

   (d) a ***contains*** r, which implies that $a = r$ since there is no region in A longer than r;

   (e) $a = r$, which implies that a ***after*** $[k, k]$.

   Thus $[k, k]$ partitions A into $B = \{a \in A | a \textit{ before } [k, k]\}$ and $C = \{a \in A | a \textit{ after } [k, k]\}$. B is a nested set with no zero-length regions on a string of length $k < n$, and C is the same kind of set on a string of length $n-k < n$. Therefore $|A| = |B|+|C| \leq (2k-1)+(2(n-k)-1) = 2n-2$.

2. $[0, n] \in A$. Then $A - \{[0, n]\}$ satisfies Case 1, so $|A| \leq 2n - 1$.

□

The $3n$ bound for nested sets is also tight. Given a string of length $n = 2^k$, form a complete binary tree whose leaves are the n characters in the string. The $2n - 1$ nodes in the tree correspond to $2n - 1$ nested regions. Adding $n + 1$ zero-length regions gives a nested set with $3n$ elements.

Theorems 1 and 2 imply that any flat, nested, or overlapping region set on a string of length n has $O(n)$ regions. Other text-processing systems take advantage of this fact by restricting all operations to flat, nested, or overlapping region sets. For example, most regular expression libraries return only a flat set of matches — finding the leftmost, longest match to the regular expression, and resuming the search for more matches *after* each match. Therefore, even though the regular expression a^* matches every region in the string $aaaa$, most regular expression libraries return only the single region $aaaa$. Similarly, Proximal Nodes [NBY95] handles only nested sets, and GC-lists [CCB95] handles only overlapping sets.

Unlike these other pattern languages, the region algebra presented in the next section makes no special distinction among flat, nested, and overlapping sets. To achieve an efficient implementation of the region algebra, however, it will be important to optimize for these kinds of sets, because they are very common in practice.

3.5 Region Algebra

This section defines the region algebra. The algebra consists of a set of operators that take zero or more region sets as arguments and produce a region set as a result.

3.5.1 Set Operators

The algebra uses the familiar operators for intersection, union, and set difference:

$$\begin{aligned} A \cap B &\equiv \{r | r \in A \wedge r \in B\} \\ A \cup B &\equiv \{r | r \in A \vee r \in B\} \\ A - B &\equiv \{r | r \in A \wedge r \notin B\} \end{aligned}$$

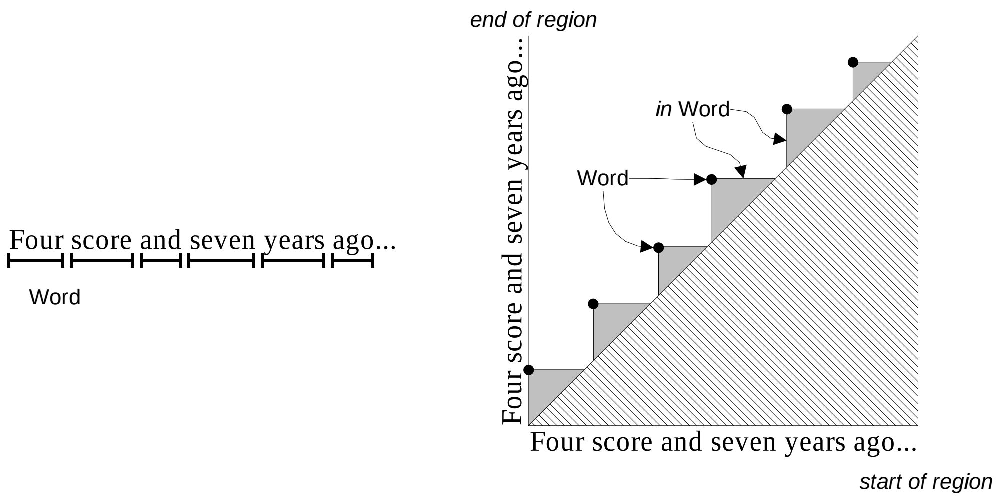

Four score and seven years ago...
Word
end of region
Four score and seven years ago...
in Word
Word
Four score and seven years ago...
start of region

Figure 3.9: The result of *in Word* depicted in region space. The isolated points make up the *Word* region set. The shaded areas are *in Word*.

In addition, the algebra defines operators for the set of all possible regions in the string and the empty set.

$$\begin{aligned} \Omega &\equiv \{[s,e] | 0 \leq s \leq e \leq n\} \\ \emptyset &\equiv \{\} \end{aligned}$$

Technically $\emptyset$ is redundant, since it could be expressed by $\Omega - \Omega$.

3.5.2 Relational Operators

The fundamental operators of the region algebra are unary operators that extend each of the fundamental region relations over a set:

$$\begin{aligned} \textit{before}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{before}\, b\} \\ \textit{after}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{after}\, b\} \\ \textit{overlaps-start}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{overlaps-start}\, b\} \\ \textit{overlaps-end}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{overlaps-end}\, b\} \\ \textit{contains}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{contains}\, b\} \\ \textit{in}\, B &\equiv \{a \in \Omega | \exists b \in B . a\, \textit{in}\, b\} \end{aligned}$$

For example, the result of applying the operator *in* to a region set B is the set of all regions that lie *in* at least one region in B.

Region space offers a handy representation for showing the results of region algebra operators. In region space, a region set corresponds to a set of points in the plane. Figure 3.9 shows the result of *in Word*, where *Word* is the set of all word regions in the string. Notice that *in Word* is just the union, across all regions w $\in$*Word*, of the region space map areas corresponding to *in* $\{w\}$. Specifically, *in Word* is the union of the triangles below and to the right of each point in *Word*.

In other systems [CC97, CCB95, GT87, NBY95], operators like *in* are defined as binary operators:

$$A\,\textit{in}\,B \quad\equiv\quad \{a \in A | \exists b \in B . a\,\textit{in}\,b\}$$

In the region algebra, relational operators are defined as unary operators instead. Since the region algebra includes operators for set intersection and for the set of all regions Ω, the binary and unary definitions are equivalent:

$$\begin{aligned} A\,\textit{in}\,B &= A \cap (\,\textit{in}\,B) \\ \textit{in}\,B &= \Omega\,\textit{in}\,B \end{aligned}$$

In this dissertation, $A\,\textit{in}\,B$ will frequently appear as syntactic shorthand for $A \cap (\,\textit{in}\,B)$.

Unary operators have some advantages. First, they offer a compact description of a *predicate*, such as $(\,\textit{before}\,A \cup \,\textit{after}\,B) \cap \,\textit{in}\,C$, without mentioning the region set to which the predicate applies. Predicates will be very useful for representing features of regions for machine learning. Second, unary operators can eliminate redundancy in some expressions. Where $(A\,\textit{before}\,B) \cup (A\,\textit{after}\,C)$ must mention A twice, the unary equivalent $A \cap (\,\textit{before}\,B \cup \,\textit{after}\,C)$ mentions A only once. When A is a complicated expression, the notational savings can be significant. Third, unary operators will simplify the implementation description presented in Chapter 4.

The set difference operator $A - B$ also has a unary equivalent $\neg B \equiv \{a \in \Omega | a \notin B\}$. Hereafter, the unary version $\neg B$ will occasionally be used as shorthand for $\Omega - B$, but the advantages of unary complement over binary difference do not seem as substantial.

3.5.3 Iteration Operator

The last operator in the algebra iterates through a region set and applies an expression to each region in the set. The iteration operator is written $\textit{forall}\,(a : A) . f(a)$, where A is a region set, a is a variable of type region set that takes on the singleton value $\{r\}$ for every region r in A, and $f(a)$ is an expression in the region algebra with a as a free variable. Formally, *forall* is defined as:

$$\textit{forall}\,(a : A) . f(a) \equiv \bigcup_{r \in A} f(\{r\})$$

Many uses of the iteration operator will be seen in the examples in the next section.

3.6 Derived Operators

To illustrate how the region algebra can be used, we now define some higher-level pattern matching operators in terms of the basic algebra. The user-level pattern language described in Chapter 6 includes many of these operators as primitives.

3.6.1 Adjacency

The intersections of the fundamental region relations correspond to boundary cases in which region endpoints touch. For example, the relation *just-before* is the intersection of the relations *before* and *overlaps-start*. Some care must be taken when applying these intersections to region sets, however. We can't simply define *just-before* as

$$\textit{just-before}\ A \equiv \textit{before}\ A \cap \textit{overlaps-start}\ A \qquad \text{(incorrect)}$$

If A contains more than one region, this definition would allow regions that are *before* one region in A and *overlaps-start* a different region in A. For example, suppose the set A consists of the two underlined regions below:

$$\texttt{Super}\underline{\texttt{cali}}\texttt{fragi}\underline{\texttt{list}}\texttt{ic}$$

Using the definition above, *just-before* A would correctly include the region `Super` because it lies *before* and *overlaps-start* the underlined region `cali`. Likewise, *just-before* A would correctly include `Supercalifragi`, which is *before* and *overlaps-start* the underlined region `list`. Yet the definition above would also match `Supercal`, because it is *before* `list` and *overlaps-start* `cali`. `Supercal` is clearly not adjacent to either of the underlined regions. In order to represent adjacency properly, we need to constrain the definition so that *before* and *overlaps-start* are only intersected when they refer to the same region in A.

The correct definition uses *forall* to iterate through the regions in A, applying the intersection to one region at a time:

$$\textit{just-before}\ A \equiv \textit{forall}\ (a : A)\ .\ \textit{before}\ a \cap \textit{overlaps-start}\ a$$

The other adjacency operators are defined similarly:

$$\begin{aligned}
\textit{just-after}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{after}\ a \cap \textit{overlaps-end}\ a\\
\textit{starts-contains}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{contains}\ a \cap \textit{overlaps-end}\ a\\
\textit{ends-contains}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{contains}\ a \cap \textit{overlaps-start}\ a\\
\textit{starts-in}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{in}\ a \cap \textit{overlaps-start}\ a\\
\textit{ends-in}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{in}\ a \cap \textit{overlaps-end}\ a
\end{aligned}$$

These relations correspond to the lines between adjacent areas in the region set map, as shown in Figure 3.10.

Another useful relation simply tests whether regions have the same start point or end point, regardless of containment:

$$\begin{aligned}
\textit{starts}\ A &\equiv \textit{starts-contains}\ A \cup \textit{starts-in}\ A\\
\textit{ends}\ A &\equiv \textit{ends-contains}\ A \cup \textit{ends-in}\ A
\end{aligned}$$

starts A is the set of all regions that start at the same place as some region in A, regardless of which region contains which. Note that it isn't necessary to use *forall* in this case, because we're taking the union of the relations, not the intersection.

Two more intersections are also of interest:

$$\begin{aligned}
\textit{start-of}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{before}\ a \cap \textit{in}\ a\\
\textit{end-of}\ A &\equiv \textit{forall}\ (a : A)\ .\ \textit{after}\ a \cap \textit{in}\ a
\end{aligned}$$

These operators always produce zero-length regions.

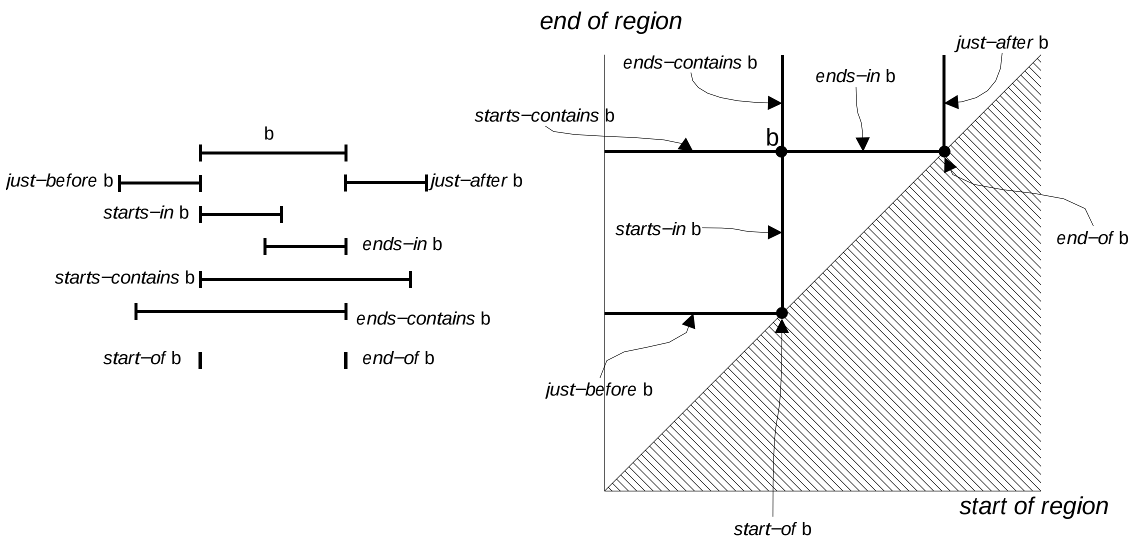


Figure 3.10: Adjacency operators.

3.6.2 Strictness

Excluding adjacency from each fundamental relation produces a set of *strict* operators:

$$\begin{aligned}
\textit{strict-before}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{before}\, a - \textit{overlaps-start}\, a \\
\textit{strict-after}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{after}\, a - \textit{overlaps-end}\, a \\
\textit{strict-in}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{in}\, a - \textit{overlaps-start}\, a - \textit{overlaps-end}\, a \\
\textit{strict-contains}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{contains}\, a - \textit{overlaps-start}\, a - \textit{overlaps-end}\, a \\
\textit{strict-overlaps-start}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{overlaps-start}\, a - \textit{before}\, a - \textit{in}\, a - \textit{contains}\, a \\
\textit{strict-overlaps-end}\, A &\equiv \textit{forall}\,(a : A)\,.\ \textit{overlaps-end}\, a - \textit{after}\, a - \textit{in}\, a - \textit{contains}\, a
\end{aligned}$$

These relations are equivalent to using strict inequalities in the definitions of the fundamental relations.

3.6.3 Overlap

Two operators are useful for describing regions that overlap:

$$\begin{aligned}
\textit{touches}\, A &\equiv \textit{forall}\,(a : A)\,.\,(\Omega - \textit{strict-before}\, a) - \textit{strict-after}\, a \\
\textit{overlaps}\, A &\equiv \textit{forall}\,(a : A)\,.\,(\Omega - \textit{before}\, a) - \textit{after}\, a
\end{aligned}$$

Intuitively, a *touches* b if a and b touch anywhere, even if only at the endpoints. Thus, for example, a *just-before* b implies a *touches* b. In contrast, the stronger relation a *overlaps* b requires that a and b have at least one character in common, if a and b are nonzero-length regions. If either a or b has zero length, then either the regions must be identical or else one region must strictly contain the other. Figure 3.11 shows the areas for *touches* and *overlaps* in region space. *overlaps* plays an important role in many algorithms in a text-processing system. For example, a display rendering algorithm has to render every region that *overlaps* the region showing in the window.

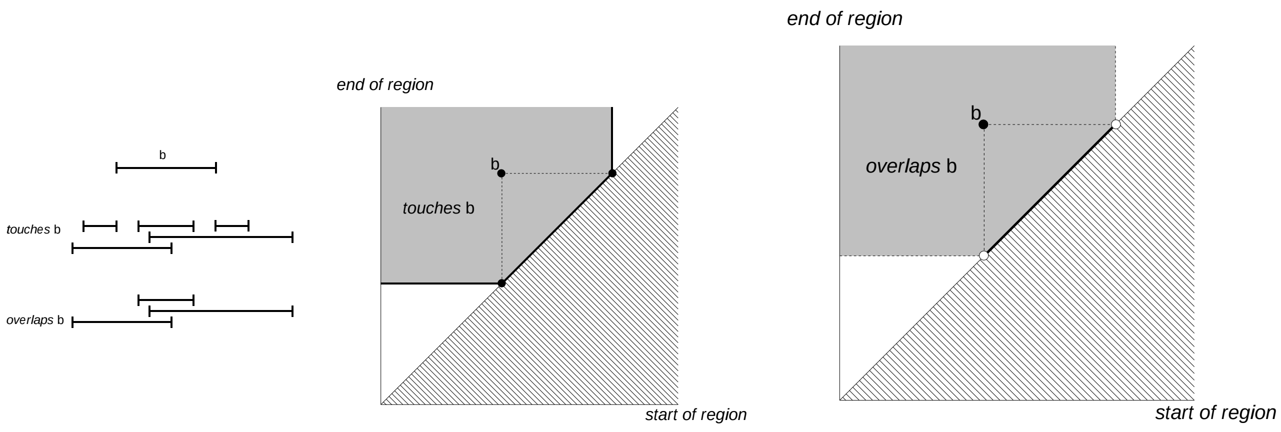


Figure 3.11: *touches* and *overlaps*.

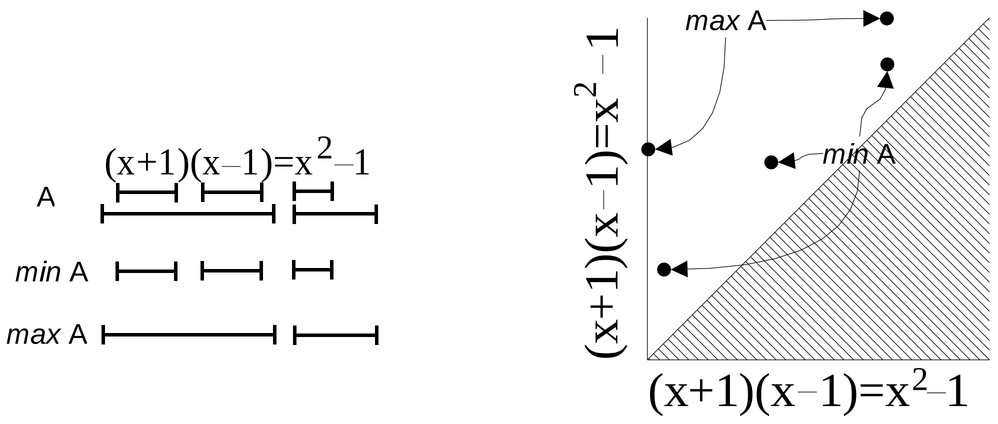


Figure 3.12: *min* and *max*.

3.6.4 Min and Max

Two operators are useful for finding the outermost and innermost regions in a region set:

$$\begin{aligned} \textit{max}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{in}\,(A - a) \\ \textit{min}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{contains}\,(A - a) \end{aligned}$$

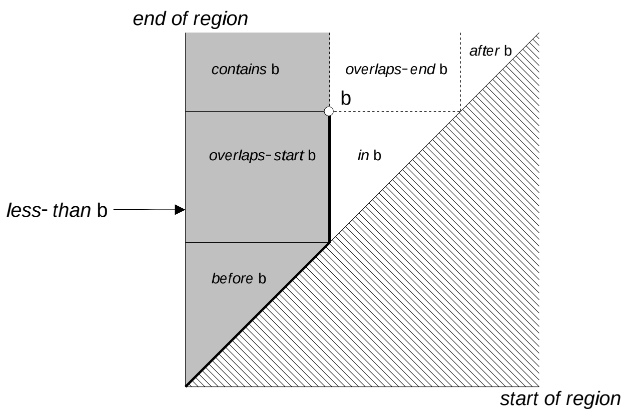
max A returns the regions in A that are not in any other region in A. Similarly, *min* A returns the regions in A that contain no other region in A. In region space, *min* A consists of the points in A closest to the diagonal, and *max* A consists of the points farthest from the diagonal (Figure 3.12).

Applied to the set of all regions Ω, *min* and *max* enable definitions of the region spanning the entire string and the set of zero-length regions:

$$\begin{aligned} \textit{entire-text} &\equiv \textit{max}\,\Omega \\ \textit{zero-length} &\equiv \textit{min}\,\Omega \end{aligned}$$

It is often useful to omit zero-length regions from a result, so the *nonzero* operator is explicitly defined for this purpose:

$$\textit{nonzero}\, A \equiv A - \textit{zero-length}$$



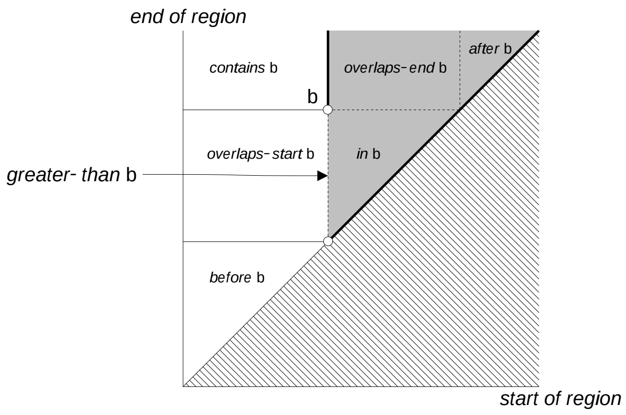


Figure 3.13: Region space areas for *less-than* and *greater-than*.

3.6.5 Counting

Counting is a common operation in structure manipulation. One may need to find the last line in a page, or the first argument of a function call.

Counting requires a total ordering on regions. We will use the conventional lexicographic ordering, so that $[s, e] < [s', e']$ if and only if either $s < s'$ or both $s = s'$ and $e < e'$. This ordering can be represented by region relations as follows:

$$\begin{aligned} \textit{less-than}\, A &\equiv \textit{forall}\,(a : A)\,.\,(\textit{before}\, a \cup \textit{overlaps-start}\, a \cup \textit{contains}\, a) - \textit{overlaps-end}\, a \\ \textit{greater-than}\, A &\equiv \textit{forall}\,(a : A)\,.\,(\textit{after}\, a \cup \textit{overlaps-end}\, a \cup \textit{in}\, a) - \textit{overlaps-start}\, a \end{aligned}$$

Figure 3.13 shows that these definitions produce the conventional lexicographic ordering, in the sense that *less-than* b corresponds to the set of points lexicographically less than point b, and *greater-than* b is the set of points lexicographically greater than b.

Now we can define *first* and *last*, a pair of operators that return the first and last region in a set by lexicographic ordering:

$$\begin{aligned} \textit{first}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{greater-than}\,(A - a) \\ \textit{last}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{less-than}\,(A - a) \end{aligned}$$

first A is the region that is not greater than any other region in A; *last* A region is the region that is not less than any other region in A. Unlike *min* and *max*, which may return more than one region, *first* and *last* return exactly one region (technically a singleton region set), as long as A is nonempty. If A is empty, then *first* and *last* return the empty set.

Using these operators, we can define a family of operators that return the nth region, counting either from the beginning or the end of the region set. The operators that count forward are defined recursively as follows:

$$\begin{aligned} \textit{nth}_{1} A &\equiv \textit{first}\, A \\ \textit{nth}_{n+1} A &\equiv \textit{first}\,(A\, \textit{greater-than}\ \textit{nth}_{n} A) \end{aligned}$$

Negative subscripts count backwards:

$$\begin{aligned} \textit{nth}_{-1} A &\equiv \textit{last}\, A \\ \textit{nth}_{-(n+1)} A &\equiv \textit{last}\,(A\, \textit{less-than}\ \textit{nth}_{-n} A) \end{aligned}$$

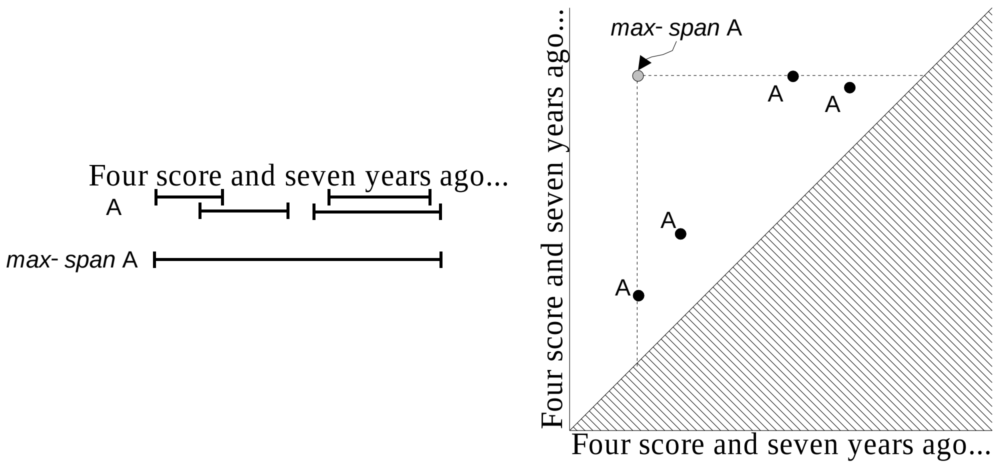

Four score and seven years ago...
A
max- span A
max- span A
A
A
A
A
Four score and seven years ago...
Four score and seven years ago...

Figure 3.14: *max-span* operator in region space.

Note that when $n > |A|$, both counting operators return the empty set.

Counting is often done relative to a context: the first word *after* a colon, or the last sentence *in* a paragraph. To represent context for counting, we can define a family of operators $\textit{nth}_n\,\textit{op}$, where *op* is a unary operator:

$$\textit{nth}_n A\ \textit{op}\ B \equiv\ \textit{forall}\,(b : B)\ .\ \textit{nth}_n(A\ \textit{op}\ b)$$

For example, $\textit{nth}_4$*Word in Sentence* returns the fourth word in every sentence.

3.6.6 Span

The *span* of two regions a and b is the region *r* such that r *starts-contains* a and r *ends-contains* b:

$$A\ \textit{span}\ B \equiv\ \textit{starts-contains}\ A \cap\ \textit{ends-contains}\ B$$

However, the span operator usually generates too many regions to be useful. The next few sections define more useful subsets of span.

The *max-span* operator returns the span of a set of regions with itself:

$$\textit{max-span}\ A \equiv\ \textit{max}\,(A\ \textit{span}\ A)$$

max-span A returns the smallest region containing every region in A, if A is nonempty; otherwise it returns the empty set. In region space, *max-span* A corresponds to the upper left corner of the smallest bounding box containing A, as shown in Figure 3.14.

3.6.7 Concatenation

The *then* operator concatenates adjacent regions:

$$A\ \textit{then}\ B \equiv\ \textit{forall}\,(a : A)\ .\ \textit{forall}\,(b : B\ \textit{just-after}\ a)\ .\ a\ \textit{span}\ b$$

This operator corresponds to the conventional definition of string concatenation. A region belongs to A *then* B if and only if it consists of a region in A concatenated with a region in B.

3.6.8 Delimiters

The *upto* operator returns the span of each region in A with the first region in B after it:

$$A \textit{ upto } B \equiv \textit{ forall}\,(a : A)\,.\ \textit{forall}\,(b :\ \textit{first}\,(B \textit{ after } a))\,.\,a \textit{ span } b$$

For example, "/*" *upto* "*/" would match C comments.

The related operator *backto* spans each region in A with the last region in B preceding it:

$$A \textit{ backto } B \equiv \textit{ forall}\,(a : A)\,.\ \textit{forall}\,(b :\ \textit{last}\,(B \textit{ before } a))\,.\,a \textit{ span } b$$

In most C programs, "*/" *backto* "/*" would match the same regions as "/*" *upto* "*/" . If some comment contained more than one occurrence of “/*”, which is permitted in C, then these expressions would return different region sets, because *backto* scans from the end delimiter.

Upto and *backto* may return overlapping sets, so neither operator is particularly useful when the start delimiter is identical to the end delimiter, as is the case for quotation marks. Suppose *QuoteMark* matches the four quotation marks in this string:

The word "zeitgeist" means "spirit of the time."

Then the expression *QuoteMark upto QuoteMark* actually matches three overlapping regions: *"zeitgeist"*, *" means "*, and *"spirit of the time."* The *backto* operator produces the same result.

Getting the region set we want, with just “zeitgeist” and “spirit of the time”, requires a left-to-right scan that alternates between starting delimiters and ending delimiters, never using the same quote mark twice. The counting operator $\textit{nth}_n$ can do such a scan, but only to a finite extent. To illustrate, suppose we had the operators *odd* and *even*, where *odd QuoteMark* returns the odd-numbered quote marks (first, third, fifth, etc.) and *even* returns the even-numbered ones. Then the expression *odd QuoteMark upto even QuoteMark* would produce the desired region set. Since *QuoteMark* is a flat set, *odd* (*QuoteMark upto QuoteMark*) would also work.

Unfortunately, defining odd and *even* in terms of $\textit{nth}_n$ requires an infinite union. *Odd* is the limit, as n goes to infinity, of the following recursive definition:

$$\begin{aligned} \textit{odd}_{\,1}A &\equiv \textit{first}\,A \\ \textit{odd}_{\,n+2} &\equiv \textit{odd}_{\,n}A \cup \textit{nth}_{\,n+2}A \end{aligned}$$

In fact, it can be shown (Chapter 5) that the region algebra with literal string matching alone cannot generate the quoted region set we want, because a region algebra expression cannot count modulo a number. Since the hypothetical *odd* and *even* operators count modulo 2, they cannot be expressed by a (finite) region algebra expression.

LAPIS solves this problem with the pattern operator *fromto*, which is defined not by an algebra expression, but by an algorithm. The FROMTO procedure (Algorithm 3.1) takes two arguments: a region set of starting delimiters L, and a region set of ending delimiters R. The procedure iterates through L and R in left-to-right order, matching a delimiter from L with the closest following delimiter from R and returning the *span* of the two delimiters. The procedure then continues with the next L delimiter after the spanned region, so that the resulting region set is always flat.

The region algebra cannot represent balanced delimiters either, where starting delimiters are matched with ending delimiters to generate a hierarchy of nested regions. Balanced parentheses

Algorithm 3.1 FROMTO returns a flat region set by spanning from each region in L to the closest following region in R.

```
FROMTO($L, R$)
1  $C \leftarrow \emptyset$
2  $l \leftarrow$ first $L$
3  $r \leftarrow$ first $R$ after $l$
4  while $r \neq \emptyset$
5  do $C \leftarrow C \cup (l$ span $r)$
6      $l \leftarrow$ first $L$ after $r$
7      $r \leftarrow$ first $R$ after $l$
8  return $C$
```

are an example of this kind of delimiter. For balanced delimiters, LAPIS introduces the *balances* operator, which is defined by the BALANCE procedure (Algorithm 3.2). Like FROMTO, BALANCE takes a set of starting delimiters L and a set of ending delimiters R. The start delimiters in L are pushed onto the stack until an end delimiter from R is encountered. Line 10 guarantees that every end delimiter used by the procedure lies *after* its matching start delimiter, every start delimiter lies *after* its parent region's start delimiter, and every end delimiter lies *after* its children's end delimiters. As a result, the output of the entire procedure is a nested region set.

Algorithm 3.2 BALANCE returns a nested region set formed by matching opening delimiters from L with closing delimiters from R.

```
BALANCE($L, R$)
 1  $C \leftarrow \emptyset$
 2  $S \leftarrow$ new STACK()
 3  $r \leftarrow$ first $(L \cup R)$
 4  while $r \neq \emptyset$
 5  do if $r \in R$ and not EMPTY($S$)
 6         then $q \leftarrow$ POP($S$)
 7              $C \leftarrow C \cup (q$ span $r)$
 8       else if $r \in L$
 9         then PUSH($S, r$)
10      $r \leftarrow$ first $(L \cup R)$ after $r$
11  return $C$
```

3.6.9 Hierarchies

Hierarchical structure is a particularly common type of text structure. Logical document structure, natural language phrase structure, and programming language syntax are all hierarchical. In most text-processing systems, hierarchical structure is represented by a *syntax tree* created by parsing the text with a grammar. In the region algebra, hierarchical structure is represented by a nested region set.

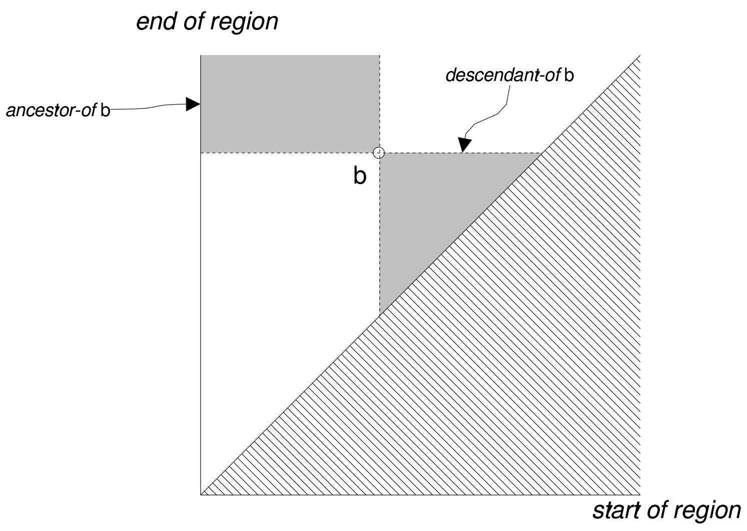


Figure 3.15: *descendant-of* and *ancestor-of*.

For capturing ancestry relations, neither *in* nor *strict-in* is quite right. For example, $a + b$ is a hierarchical structure of three expressions: a, b, and $a+b$. We would like to express the relationship that a and b are descendants of $a + b$. Both a and b are *in* $a + b$, but so is $a + b$ itself, so *in* does not describe the descendant relation. Yet neither a nor b is *strict-in* $a + b$, because the start of a (end of b) coincides with the corresponding point of $a + b$. This problem can be fixed by defining a new pair of operators:

$$\begin{aligned} \textit{descendant-of}\, B &\equiv \textit{forall}\,(b : B)\,.\,(\,\textit{in}\, b) - b \\ \textit{ancestor-of}\, B &\equiv \textit{forall}\,(b : B)\,.\,(\,\textit{contains}\, b) - b \end{aligned}$$

In region space, these operators correspond to *in* and *contains* with a corner removed, as shown in Figure 3.15. Note that *descendant-of* and *ancestor-of* do not specify the hierarchy of interest. They are defined as unary predicates, so that *ancestor-of* b matches *any* region that could be an ancestor of b. The hierarchy of interest is specified by intersection. For example, *Statement* $\cap$ *ancestor-of* b returns the statements that are ancestors of b, *Method* $\cap$ *ancestor-of* b returns the method definitions, and *Line* $\cap$ *ancestor-of* b returns the lines. Thus, each use of a hierarchy operator can refer to a different hierarchy. This is an important difference from other systems, which support one and only one syntax tree. Furthermore, b need not be an exact member of the hierarchy in question — it need not correspond exactly to a piece of Java syntax, for example, or to a line. A sloppy selection made by the user, or a partial literal string match, is a sufficient entry point into a hierarchy.

Applying *min* and *max* to the hierarchy operators, one obtains the parent, child, root, and leaf relationships:

$$\begin{aligned} A\,\textit{child-of}\, B &\equiv \textit{max}\,(A\,\textit{descendant-of}\, B) \\ A\,\textit{leaf-of}\, B &\equiv \textit{min}\,(A\,\textit{descendant-of}\, B) \\ A\,\textit{parent-of}\, B &\equiv \textit{min}\,(A\,\textit{ancestor-of}\, B) \\ A\,\textit{root-of}\, B &\equiv \textit{max}\,(A\,\textit{ancestor-of}\, B) \end{aligned}$$

Unlike *descendant-of* and *ancestor-of*, these operators must be binary. The hierarchy must be specified in order to find the *min* and *max* of ancestors and descendants.

3.6.10 Before/After with Restricted Range

One drawback of *before* and *after* as pattern-matching operators is that they match too many regions. *before* B matches anywhere in the string as long as it lies before at least one region in B. This means that all but the last region in B is irrelevant; *before* B is equivalent to *before last* B.

Frequently we want to restrict the range of the search to some region set C:

$$\begin{aligned}\textit{before-in}\,(B,C) &\equiv \textit{forall}\,(c:C)\,.\,\textit{forall}\,(b:B\,\textit{in}\,c)\,.\,\textit{before}\,b\cap\,\textit{in}\,c\\ \textit{after-in}\,(B,C) &\equiv \textit{forall}\,(c:C)\,.\,\textit{forall}\,(b:B\,\textit{in}\,c)\,.\,\textit{after}\,b\cap\,\textit{in}\,c\end{aligned}$$

before-in (B,C) matches any region that is before some B region but still in the same C region. For example, *before-in* ("@" , *EmailAddress*) would match any region preceding the “@” in an email address.

Other structured text query languages [NBY95, KM98] restrict *before* and *after* even further to return only the closest match before or after each region in B. This concept can be represented by applying *first* and *last*. For example, here is how the Proximal Nodes [NBY95] syntax and semantics for *before* and *after* would be expressed in the region algebra:

$$\begin{aligned}A\,\textit{before}\,B\,(C) &\equiv \textit{last}\,(A\,\textit{before-in}\,(B,C))\\ A\,\textit{after}\,B\,(C) &\equiv \textit{first}\,(A\,\textit{after-in}\,(B,C))\end{aligned}$$

3.6.11 Split

Awk [AKW88], Perl [WCS96] and other text-processing languages include a function for splitting a string into parts. In awk, for example, every line is split automatically into fields by a delimiter pattern, which is whitespace by default. Awk also provides the `split` function to split any string into pieces separated by a delimiter pattern. The standard C library includes `strtok`, which splits a string into pieces separated by one or more delimiter characters.

Splitting can be represented in the region algebra as follows. Suppose D is a set of delimiter regions. For convenience, we’ll name the complement of *overlaps*:

$$\textit{separated-by}\,D \equiv \Omega - \textit{overlaps}\,D$$

separated-by D matches any region that doesn’t overlap a delimiter in D. Figure 3.16 shows the region space areas. We can split the entire string around the delimiters D by applying *max* to the result of *separated-by*:

$$\textit{split}\,D \equiv \textit{max}\,(\,\textit{separated-by}\,D)$$

It is also useful to split another set of regions by the delimiters:

$$A\,\textit{split}\,D \equiv \textit{forall}\,(a:A)\,.\,\textit{max}\,(\,\textit{in}\,a\cap\,\textit{separated-by}\,D)$$

For example, awk’s behavior of splitting lines into fields can be obtained with the expression *Line split Whitespace*, where Whitespace matches a contiguous run of space and tab characters.

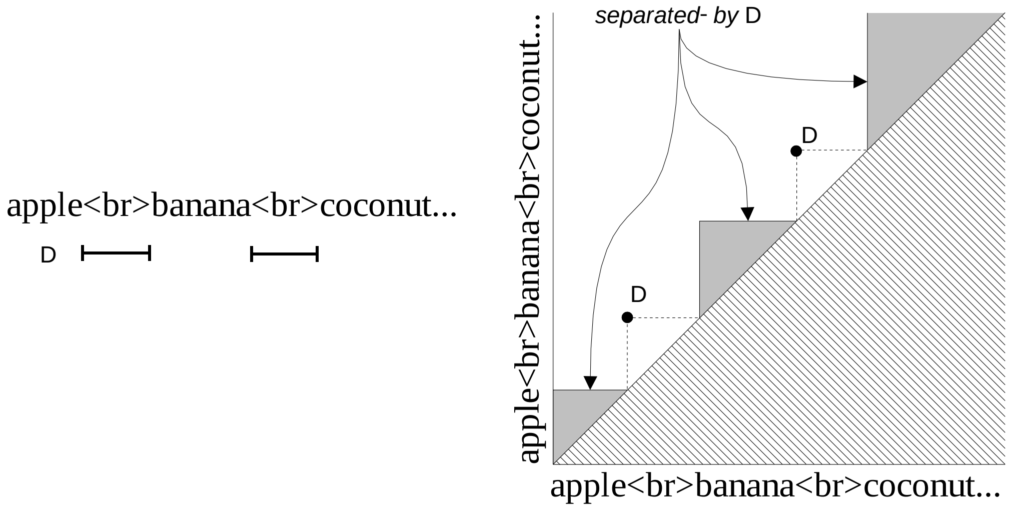

apple<br>banana<br>coconut...
D
separated- by D
D
D
apple<br>banana<br>coconut...
apple<br>banana<br>coconut...

Figure 3.16: *separated-by* D.

When two delimiters are adjacent, *split* returns the zero-length region between them. The `split` functions in Perl and awk follow the same semantics. The `strtok` function in ANSI C never returns empty strings, skipping over multiple adjacent delimiters instead. A simple way to describe this alternative semantics uses the *nonzero* operator to remove all zero-length regions from the result:

$$\textit{split-nonzero}\, D \equiv \textit{nonzero}\,(\textit{split}\, D)$$

A more subtle definition takes advantage of the fact that *overlaps* D is always a superset of D. Thus its complement, *separated-by* D, never includes an element of D, so *split* D = *max* (*separated-by* D) cannot include a delimiter either. If we enlarge the set of delimiters to include the (zero-length) start points and end points of every delimiter in D, then *split* will not return any zero-length regions. Applying the *in* operator is a simple way to do this, giving the equivalent definition

$$\textit{split-nonzero}\, D = \textit{split in}\, D$$

Figure 3.17 shows how *split* and *split-nonzero* differ.

3.6.12 Flatten

Some text manipulations require a flat region set. For example, deleting or sorting a set of regions in a string is complicated if regions can nest or overlap. For such tasks, it makes sense to convert the set into a flat region set first.

Nested region sets can be flattened by applying *max* or *min* . The former operator returns the roots of the hierarchy, and the latter returns the leaves. Intermediate levels can be obtained by applying *child-of* or *parent-of*. For example, *max* (A *child-of* A) returns all the children of the roots.

An arbitrary region set can be flattened by replacing each group of overlapping regions with the region spanning the group. This operator, *flatten*, can be described with two applications of *split* . *split* A matches regions that span the gaps of A — characters that are not covered by any region in A. So *split split* A matches the regions between the gaps. The only problem with *split split* A

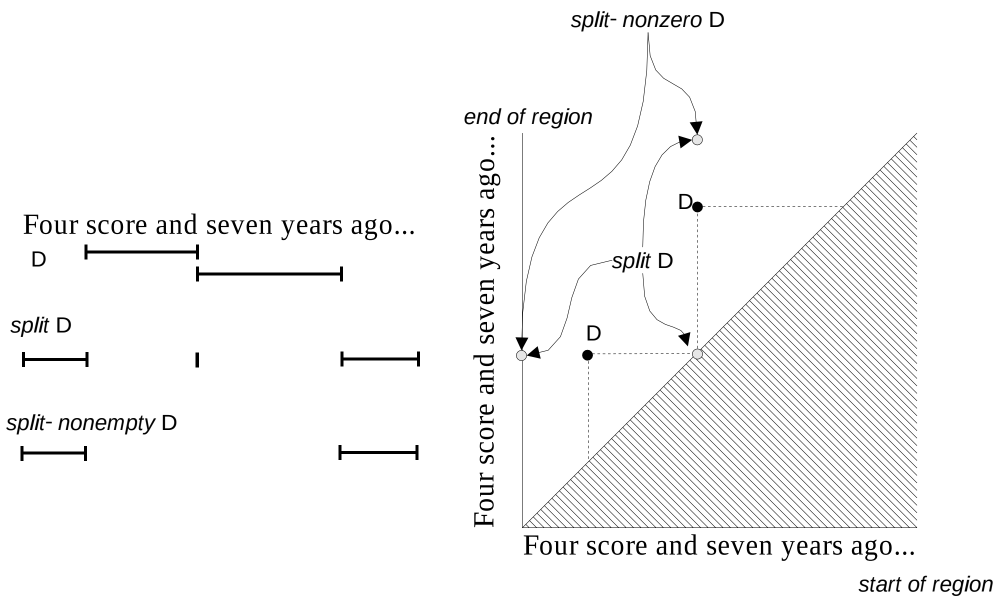

split- nonzero D
end of region
Four score and seven years ago...
D
split D
split- nonempty D
D
split D
D
Four score and seven years ago...
Four score and seven years ago...
start of region

Figure 3.17: *split* and *split-nonzero*.

is that it always includes the zero-length regions that start and end the entire string. We have to adjust the definition of *flatten* to exclude those two points if A lacks them:

$$\textit{flatten}\, A \equiv (\textit{split split}\, A) - ((\textit{start-of entire-text} \cup \textit{end-of entire-text}) - A)$$

If two regions in A are merely adjacent, not overlapping, then *flatten* A does not combine them, because *split* A leaves a zero-length region between the two regions that causes *split split* A to return the two regions separately. It is sometimes useful to coalesce adjacent regions as well as overlapping regions. To do that, we just replace the innermost application of *split* with *split-nonempty*:

$$\textit{melt}\, A \equiv (\textit{split split-nonempty}\, A) - ((\textit{start-of entire-text} \cup \textit{end-of entire-text}) - A)$$

The difference between *flatten* and *melt* can be seen in Figure 3.18.

3.6.13 Ignoring Background

It is often useful to weaken the adjacency test in such a way that two regions are considered adjacent as long as they are only separated by irrelevant characters. Call these irrelevant characters the *background*. The appropriate background depends on the pattern and the string to which it is applied. When searching for phrases in natural language, for example, a good background would be whitespace and punctuation. When searching for statements or expressions in source code, the background might be whitespace and comments. When searching an HTML or XML document, a good background might be whitespace and tags. Background can make many pattern expressions simpler.

The adjacency operators described in Section 3.6.1 can be extended to include a background parameter, which is just a set of regions W. Recalling the definition of *just-before* and *just-after*,

$$\begin{aligned} \textit{just-before}\, A &\equiv \textit{forall}\,(a : A) \,.\, \textit{before}\, a \cap \textit{overlaps-start}\, a \\ \textit{just-after}\, A &\equiv \textit{forall}\,(a : A) \,.\, \textit{after}\, a \cap \textit{overlaps-start}\, a \end{aligned}$$

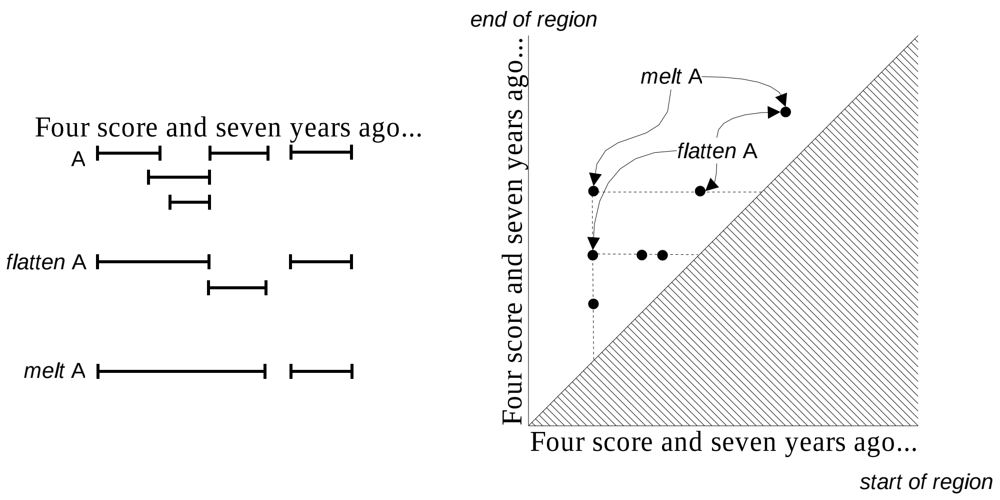

Four score and seven years ago...
A
flatten A
melt A
end of region
start of region
Four score and seven years ago...
Four score and seven years ago...
melt A
flatten A

Figure 3.18: *flatten* creates a flat set by combining overlapping regions into a single region. *melt* combines not only overlapping regions but also adjacent regions.

we can extend these definitions to ignore at most one intervening occurrence of a region from W:

$$\begin{aligned}\textit{just-before}_W A &\equiv \textit{forall}\,(a : A)\,.\ \textit{before}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-start}\,a))\\ \textit{just-after}_W A &\equiv \textit{forall}\,(a : A)\,.\ \textit{after}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-start}\,a))\end{aligned}$$

To allow an arbitrary number of intervening or partial background regions, we can melt the background regions (i.e. use *melt* W as the background parameter). Figure 3.19 illustrates *just-before* $_W$ and *just-after* $_W$.

Similar background-sensitive definitions can be given for other adjacency operators:

$$\begin{aligned}\textit{starts-contains}_W A &\equiv \textit{forall}\,(a : A)\,.\ \textit{contains}\,a \cap (\textit{overlaps-end}\,a \cup \textit{overlaps-end}\,(W\ \textit{overlaps-start}\,a))\\ \textit{ends-contains}_W A &\equiv \textit{forall}\,(a : A)\,.\ \textit{contains}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-end}\,a))\\ \textit{starts-in}_W A &\equiv \textit{forall}\,(a : A)\,.\ \textit{in}\,a \cap (\textit{overlaps-start}\,a \cup \textit{overlaps-end}\,(W\ \textit{overlaps-start}\,a))\\ \textit{ends-in}_W A &\equiv \textit{forall}\,(a : A)\,.\ \textit{in}\,a \cap (\textit{overlaps-end}\,a \cup \textit{overlaps-start}\,(W\ \textit{overlaps-end}\,a))\end{aligned}$$

Operators defined in terms of the adjacency operators can also be made background-sensitive:

$$\begin{aligned}\textit{starts}_W A &\equiv \textit{starts-contains}_W A \cup \textit{starts-in}_W A\\ \textit{ends}_W A &\equiv \textit{ends-contains}_W A \cup \textit{ends-in}_W A\end{aligned}$$

Background is also helpful in operators that depend on adjacency, such as *then*. Here is a background-sensitive version of *then*:

$$A\ \textit{then}_W B \equiv \textit{forall}\,(a : A)\,.\ \textit{forall}\,(b : B\ \textit{just-after}_W a)\,.\ a\ \textit{span}\ b$$

Background can also be used to weaken set intersection, so that two regions are considered equal if one can be obtained from the other by trimming or adding background characters. We start by defining $\textit{equals}\,A \equiv \textit{forall}\,(a : A)\,.\ \textit{starts}\,a \cap \textit{ends}\,a$, which is just the identity function.

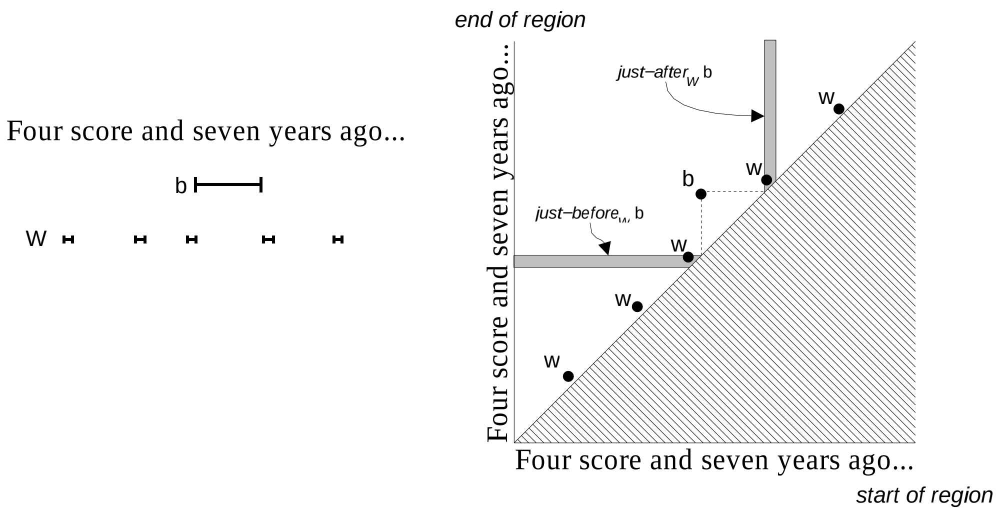

Four score and seven years ago...
b
W
end of region
Four score and seven years ago...
just–after$_W$ b
just–before$_W$ b
w
b
Four score and seven years ago...
start of region

Figure 3.19: *just-before*$_W$ and *just-after*$_W$.

But using the background-sensitive versions of *starts* and *ends* produces a background-sensitive equality test:

$$\textit{equals}_W A \equiv \textit{forall}\,(a : A)\,.\ \textit{starts}_W a \cap \textit{ends}_W a$$

Figure 3.20 shows the regions produced by *equals*$_W A$.

A pattern would use A *equals*$_W B$ instead of $A \cap B$ in order to allow weak equality between regions in A and B. Weak equality is particularly useful for comparing regions from different hierarchies, such as physical and logical layout, or regions produced by different parsers. Weak equality also improves usability. For example, weak equality allows a user's selection to match a piece of Java syntax even if it does not start and end at the precise character positions identified by the Java parser, as long as the differences are limited to background characters.

3.6.14 Trim

Many string manipulation libraries include functions that remove whitespace from the start or end of a string. One example is the `trim` method in `java.lang.String`, which trims whitespace from both ends of the string. Perl also includes `chomp`, which removes a linebreak from the end of a string. We can generalize these routines with a region algebra expression which trims the ends of a set of regions.

Trim-left and *trim-right* are binary operators that trim the left or right end of each region in A to remove at most one overlapping occurrence of a "whitespace" region from W. To trim an arbitrary number of whitespace regions, just use a melted region set for W. The trim operators are defined as follows:

$$\begin{aligned} A\,\textit{trim-right}\,W &\equiv \textit{forall}\,(a : A)\,.\,(\textit{start-of}\,a)\ \textit{upto}\ (\textit{end-of}\,a \cup W\ \textit{overlaps-end}\,a) \\ A\,\textit{trim-left}\,W &\equiv \textit{forall}\,(a : A)\,.\,(\textit{end-of}\,a)\ \textit{backto}\ (\textit{start-of}\,a \cup W\ \textit{overlaps-start}\,a) \end{aligned}$$

For example, the effect of Perl's `chomp` can be obtained by *Line trim-right Linebreak*.

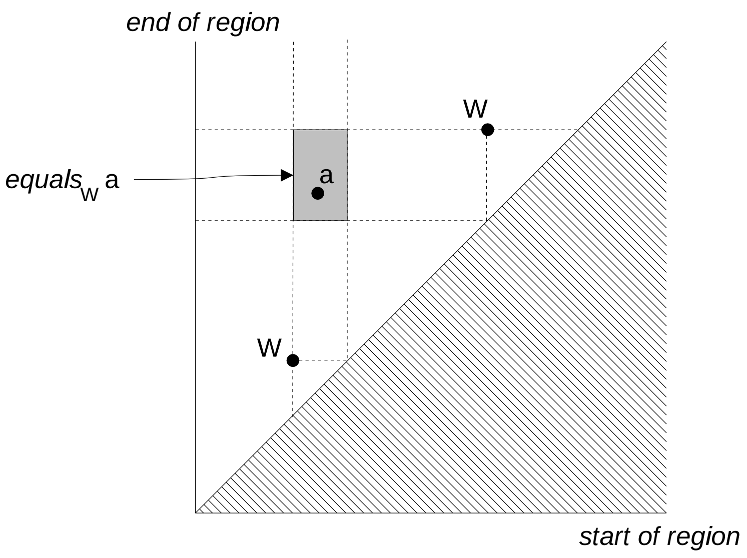


Figure 3.20: $equals_W a$ matches any region that can be obtained from a by adding or removing characters in W.

Trim applies both *trim-left* and *trim-right:*

$$A\ \textit{trim}\ W \equiv (A\ \textit{trim-right}\ W)\ \textit{trim-left}\ W$$

So the `trim` method of `java.lang.String` can be obtained by *trim Whitespace*.

Chapter 4

Region Algebra Implementation

This chapter describes how the region algebra is implemented in LAPIS. The key aspect of the implementation is the representation chosen for region sets, which has two parts:

- A *region rectangle* is a rectangle in region space. Region rectangles are a compact way to represent the result of applying a relational operator to a region.
- A *rectangle collection* represents a region set as a union of region rectangles. We can draw on research in computational geometry to find efficient data structures for rectangle collections.

Section 4.6 describes how the basic implementation can be optimized to significantly improve its performance in special cases that are common in practice.

4.1 Region Rectangles

A *region rectangle* is a tuple of four integers, (s_1, e_1, s_2, e_2), which represents a set of regions as a closed rectangle in region space:

$$(s_1, e_1, s_2, e_2) \equiv \{[s, e] | s_1 \leq s \leq s_2 \wedge e_1 \leq e \leq e_2\}$$

Essentially, a region rectangle is a set of regions whose start point and end point must fall into the specified half-open interval. Figure 4.1 shows some region rectangles corresponding to region sets in a text string.

A few facts about region rectangles follow immediately from the definition:

- The single region $[s, e]$ corresponds to the region rectangle (s, e, s, e).
- The set of all possible regions Ω in a string of length n is the region rectangle $(0, 0, n, n)$.
- One region rectangle is a subset of another, $(s_1, e_1, s_2, e_2) \subseteq (s'_1, e'_1, s'_2, e'_2)$, if and only if $s'_1 \leq s_1 \leq s_2 \leq s'_2$ and $e'_1 \leq e_1 \leq e_2 \leq e'_2$.
- One region rectangle intersects another, $(s_1, e_1, s_2, e_2) \cap (s'_1, e'_1, s'_2, e'_2) \neq \emptyset$, if and only if $s_1 \leq s'_2$, $s'_1 \leq s_2$, $e_1 \leq e'_2$, and $e'_1 \leq e_2$.

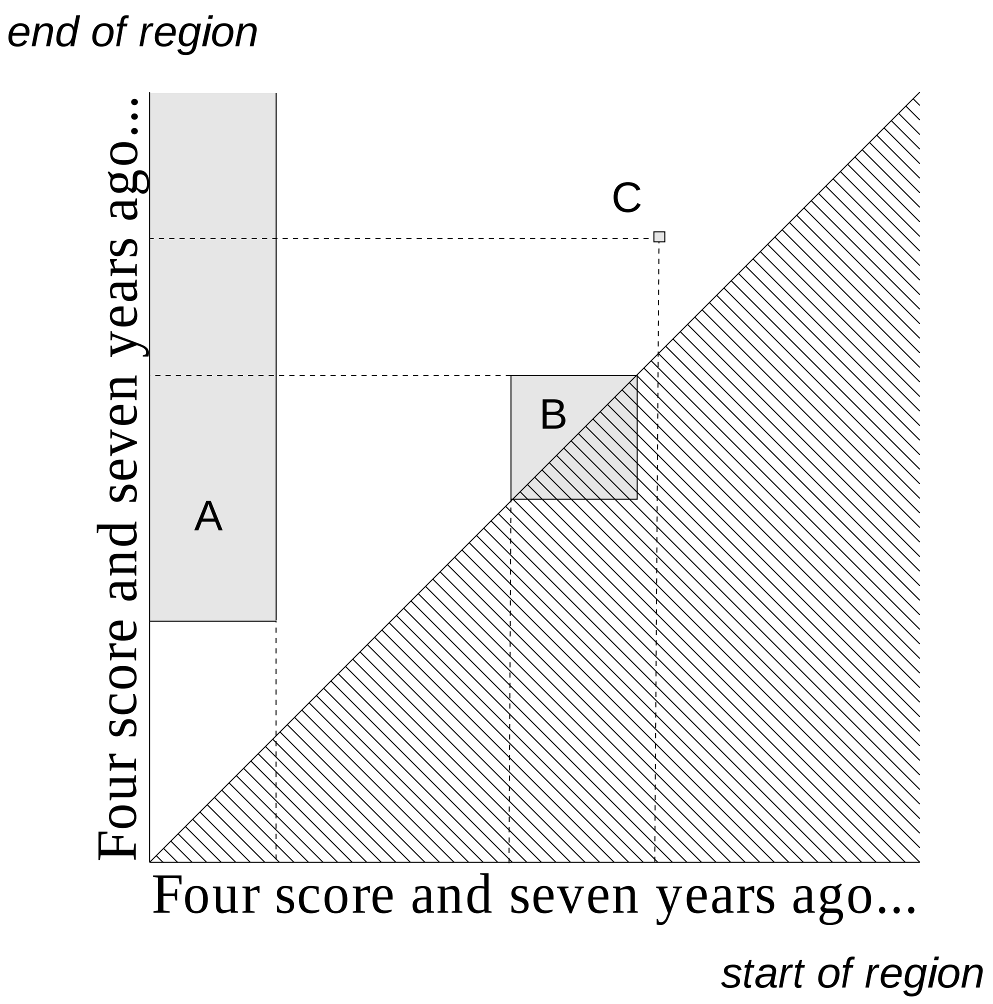
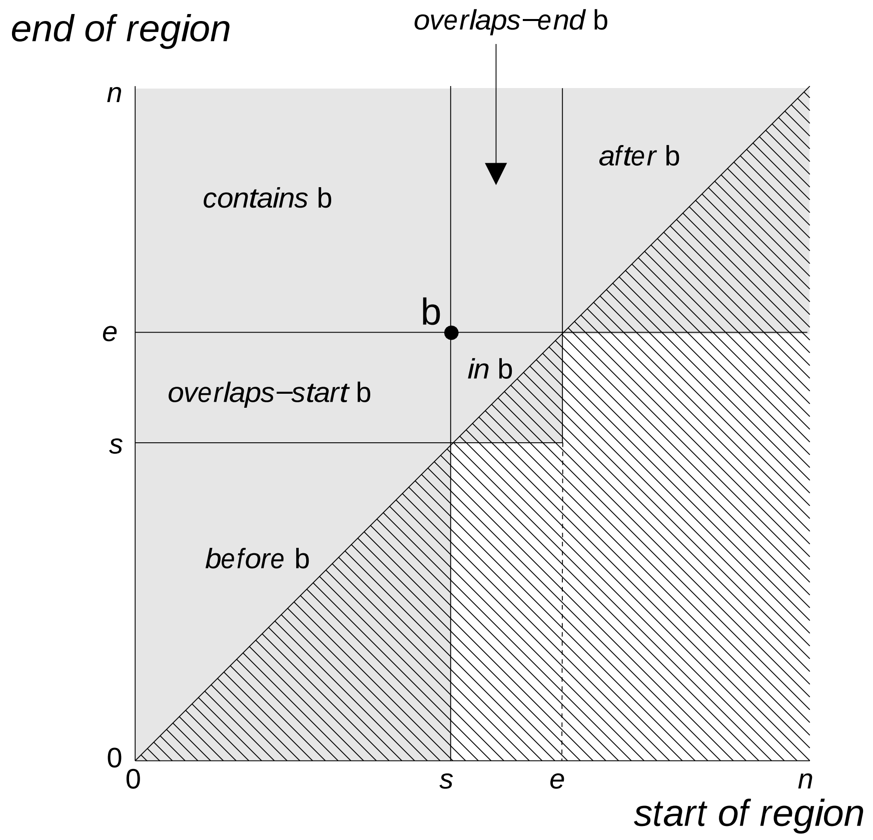

end of region
Four score and seven years ago...
C
B
A
Four score and seven years ago...
start of region

Figure 4.1: Region rectangles depicted in a text string and in region space. Region rectangle A is the set of all regions that *contains* the word “score”, B is the set of regions that are *in* the word “seven”, and C consists of the single region “years”.



$$b = [s,e]$$

$$\begin{aligned} \textit{before}\, b &= (0,0,s,s) & \textit{after}\, b &= (e,e,n,n) \\ \textit{overlaps-start}\, b &= (0,s,s,e) & \textit{overlaps-end}\, b &= (s,e,e,n) \\ \textit{contains}\, b &= (0,e,s,n) & \textit{in}\, b &= (s,s,e,e) \end{aligned}$$

Figure 4.2: Region relations can be represented by rectangles.

A region rectangle is capable of representing certain region sets very compactly. With only four integers, a region rectangle can describe a set as large as Ω. One particularly interesting kind of region set can be represented with a region rectangle: the result of a region relation operator. Recall from Figure 3.6 that a region space map shows how every region in region space is related to a given region b. The map has areas for all the fundamental relations: *before* b, *overlaps-start* b, *contains* b, *in* b, *overlaps-end* b, and *after* b. Each of these areas is the result of applying an algebraic operator to the set $\{b\}$.

The key insight is that every area in b's region space map can be represented by a rectangle, as shown in figure 4.2. Some areas are already rectangular (*contains* b, *overlaps-start* b, and *overlaps-end* b). Other areas (*before* b, *in* b, and *after* b) are triangular, cut by the $45°$ diagonal. But even these areas can be represented by a rectangular area, part of which extends below the diagonal, as long as the part below the diagonal is implicitly ignored. Thus, the result of applying any of the six region relation operators to a region $[s,e]$ can always be represented by one rectangle.

We can go beyond single regions $[s,e]$, however. Applying a relational operator to any region rectangle (s_1, e_1, s_2, e_2) will also produce a rectangle. In other words:

Claim 2. *The set of region rectangles is closed under the relational operators* before, after, overlaps-start, overlaps-end, in, *and* contains.

Figure 4.3 demonstrates a geometric proof of Claim 2.

This closure property is the reason why region rectangles are so useful as a fundamental representation. Thanks to the closure property, if a region set A is represented by N rectangles, then *op* A can also be represented by N rectangles, for any relational operator *op*. The closure property also extends to the derived relational operators that were defined in Chapter 3, including *just-before*, *just-after*, *starting*, *ending*, and *overlaps*. A few of these operators are illustrated in Figure 4.3.

Region rectangles are also closed under set intersection, since the intersection of two rectangles is also a rectangle. Region rectangles are not closed under union or difference, however.

4.2 Rectangle Collection

Any region set can be represented by a union of region rectangles. Some examples of rectangle collections for typical region sets are shown in Figures 4.4–4.6.

A *rectangle collection* is the fundamental representation for a region set in LAPIS. For now, we will treat a rectangle collection as an abstract data type representing a set of points in region space. Points can be queried, added, and removed as axis-aligned rectangles with integer coordinates. A rectangle collection C supports four operations:

- QUERY(C, r) searches C for all points that lie in the query rectangle r. The result is a stream of rectangles, not necessarily disjoint, so the same point can be returned more than once in different rectangles. All points in the rectangle collection are enumerated by QUERY(C, Ω).

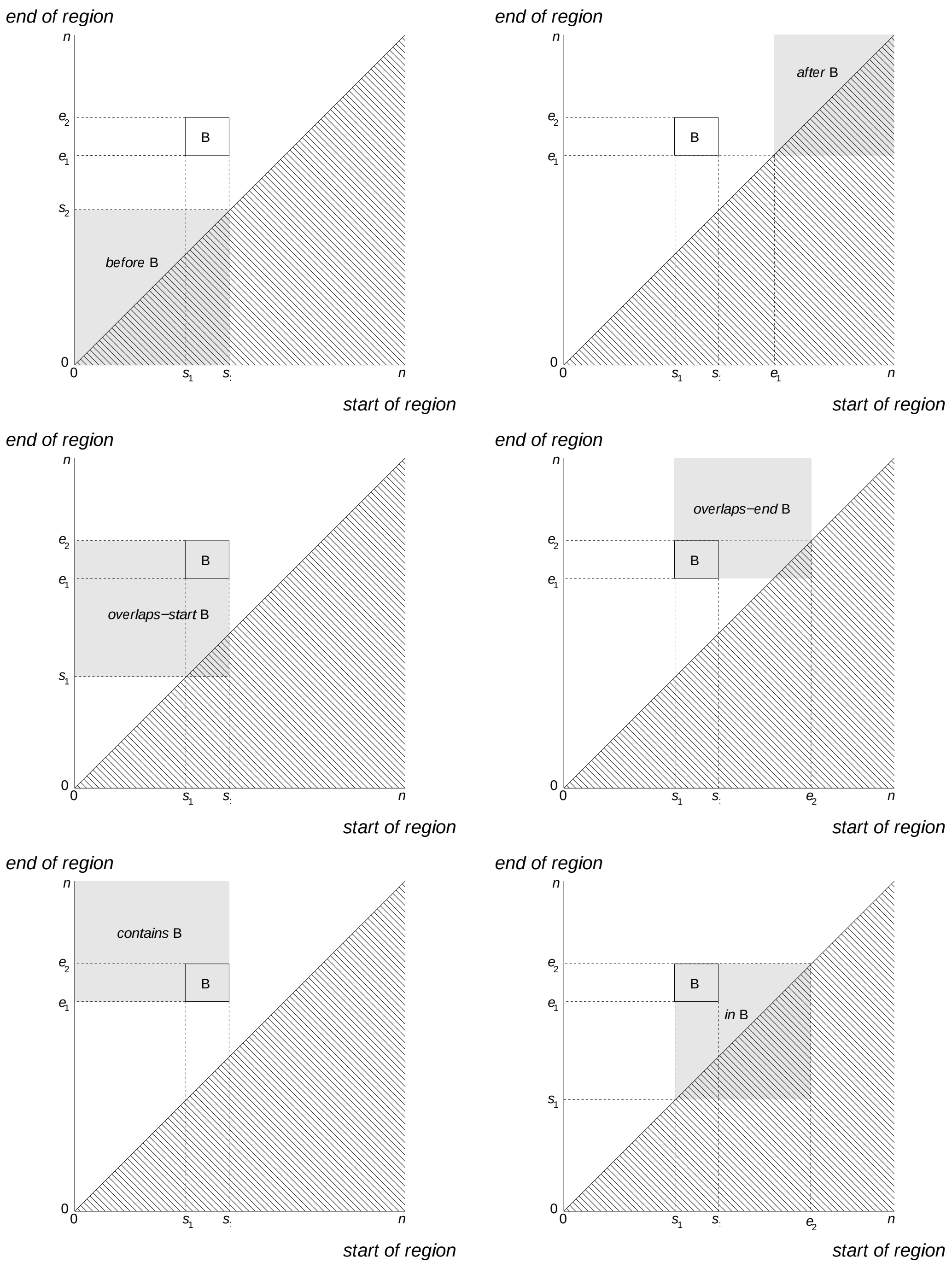
- INSERT(C, r) inserts a rectangle r into C, so that the set of points represented by C now includes all the points in r.

- DELETE(C, r) deletes a rectangle r from C, so that the set of points represented by C no longer includes any of the points in r.

- COPY(C) duplicates a rectangle collection, hopefully faster than enumerating its contents and inserting them into an empty collection.

These operations are extensions of the conventional membership test, insert, and delete operations for sets. Instead of taking a single element to query, insert, or delete, the operations take a region rectangle, a set of related elements.

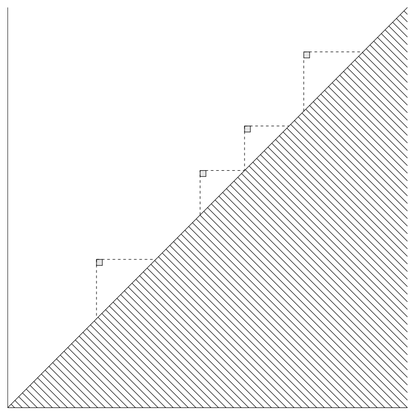
A rectangle collection only needs to represent the region set faithfully, not the particular set of rectangles that were inserted to create it. A rectangle collection can merge rectangles, split rectangles, and throw away redundant rectangles, in order to store the collection more compactly or make queries faster. As a result, QUERY(C, Ω) need not return the same set of rectangles that were inserted into the collection, as long the union of the rectangles returned by QUERY is identical to the union of the inserted rectangles.



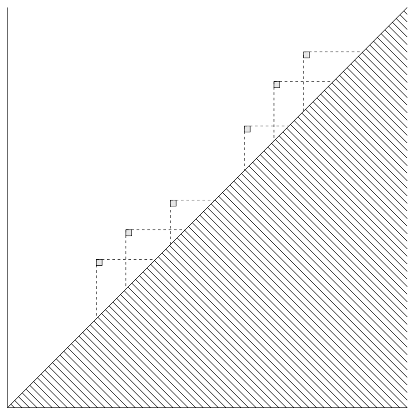
$$B = (s_1, e_1, s_2, e_2)$$

$$\begin{aligned}
\textit{before}\, B &= (0, 0, s_2, s_2) & \textit{after}\, B &= (e_1, e_1, n, n)\\
\textit{overlaps-start}\, B &= (0, s_1, s_2, e_2) & \textit{overlaps-end}\, B &= (s_1, e_1, e_2, n)\\
\textit{contains}\, B &= (0, e_1, s_2, n) & \textit{in}\, B &= (s_1, s_1, e_2, e_2)
\end{aligned}$$

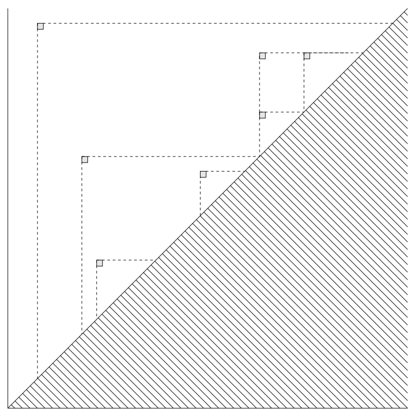
Figure 4.3: Applying a relational operator to a region rectangle always produces another rectangle.



flat region set *F*



overlapping region set *O*



nested region set *N*

Figure 4.4: Rectangle collections for flat, overlapping, and nested region sets. Each rectangle covers only a single point. Dashed lines show that each region set meets the desired definition. Regions in F must be *before* or *after* each other. Regions in O may also *overlap-start* or *overlap-end*. Regions in N may be related by *before, after, in,* or *contains.*

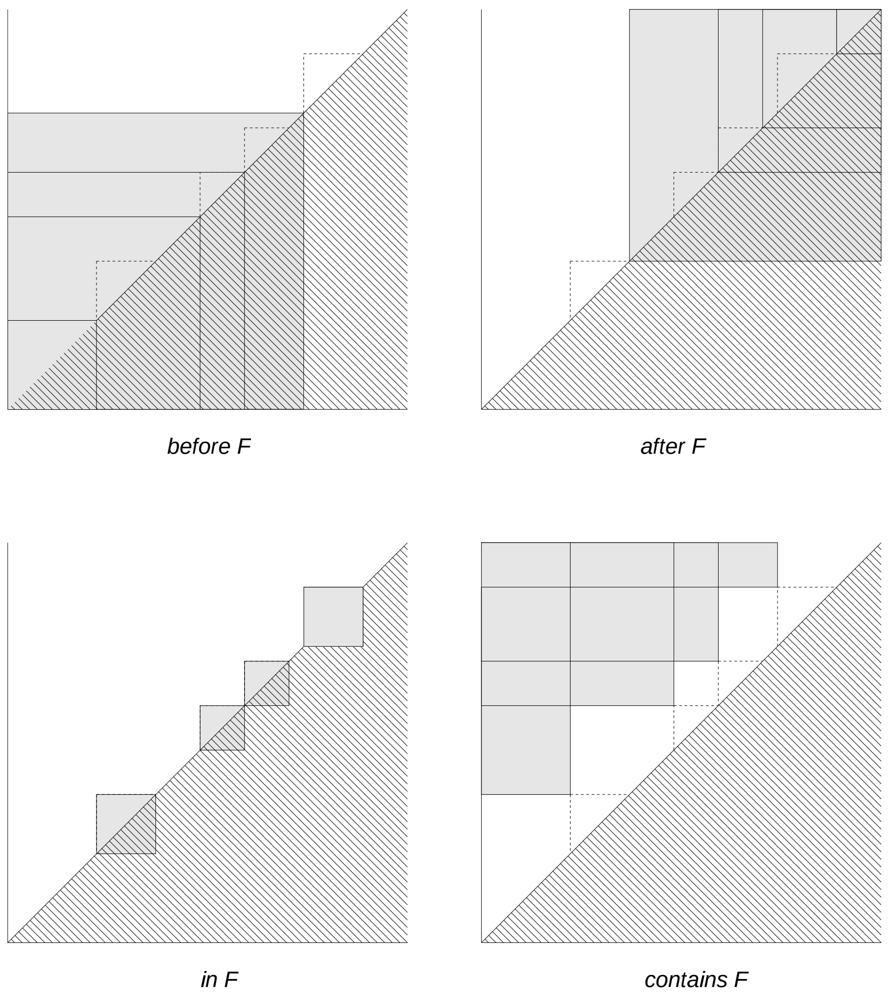


Figure 4.5: Rectangle collections for relational operators applied to the flat region set F from Figure 4.4. Each collection contains four rectangles, which may intersect. Dashed lines show the location of the original region set F.

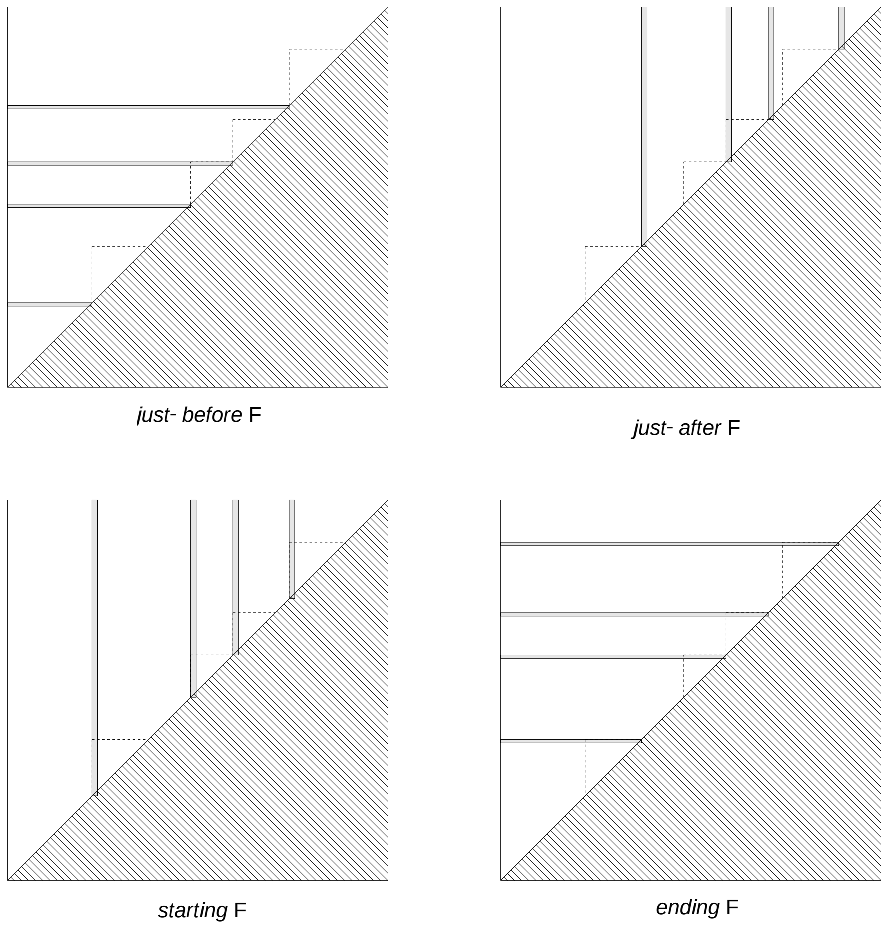


Figure 4.6: More relational operators applied to the flat region set F from Figure 4.4. Dashed lines show the location of the original region set F.

4.3 Implementing the Region Algebra

Now we are ready to implement the algebra operators using rectangle collections. Recall that the region algebra described in Section 3.5 has the following operators:

- Relational operators: *before, after, in, contains, overlaps-start, overlaps-end*.
- Set operators: $\cup$, $\cap$, $-$.
- Iteration operator: *forall*.

The relational operators are implemented by Algorithm 4.1. The key line of the algorithm is line 3, in which the relational operator *op* is applied to a rectangle, producing another rectangle which is then inserted into the result. Figure 4.3 shows the rectangle produced by each relational operator.

Algorithm 4.1 RELATION applies a relational operator to a rectangle collection.

```
RELATION(op, A)
1  U ← new RECTANGLECOLLECTION
2  for each r in A
3  do INSERT(U, op r)
4  return U
```

Set union, intersection, and difference are implemented by Algorithms 4.2–4.4. Since union and intersection are commutative, A and B can be passed in any order. UNION and INTERSECTION are written under the assumption that $|A| > |B|$, in the sense that enumerating the rectangles in A takes longer than enumerating the rectangles in B. Thus UNION copies A instead of enumerating it, and INTERSECTION queries A instead of enumerating it. If the caller can cheaply determine which collection is larger, then this optimization may save some time.

Algorithm 4.2 UNION finds the union of two rectangle collections.

```
UNION(A, B)
1  U ← COPY(A)
2  for each r in B
3  do INSERT(U, r)
4  return U
```

Finally, the *forall* operator is shown in Algorithm 4.5. FORALL iterates through all the rectangles in a rectangle collection and applies a function f to every rectangle. Each application of f returns a stream of rectangles, which are inserted into a new rectangle collection and returned. The function f takes a rectangle (s_1, e_1, s_2, e_2) and produces a stream of rectangles that result from applying the body of the *forall* expression to every region $[s, e]$ in the rectangle. Although in general this may require iterating through all pairs $[s, e]$ such that $s_1 \le s < s_2$ and $e_1 \le e < e_2$, in practice f can compute a result for the entire rectangle at once. All the uses of *forall* in the examples in Chapter 3 behave this way.

Algorithm 4.3 INTERSECTION finds the intersection of two rectangle collections.

```
INTERSECTION($(A, B)$)
1  $U \leftarrow$ **new** RECTANGLECOLLECTION
2  **for each** $r$ **in** $B$
3  **do for each** $r'$ **in** QUERY$(A, r)$
4         **do** INSERT$(U, r')$
5  **return** $U$
```

Algorithm 4.4 DIFFERENCE finds the difference between two rectangle collections.

```
DIFFERENCE$(A, B)$
1  $U \leftarrow$ COPY$(A)$
2  **for each** $r$ **in** $B$
3  **do** DELETE$(U, r)$
4  **return** $U$
```

Algorithm 4.5 FORALL applies a function f to every rectangle in collection C.

```
FORALL$(A, f)$
1  $U \leftarrow$ **new** RECTANGLECOLLECTION
2  **for each** $r$ **in** $A$
3  **do for each** $r'$ **in** $f(r)$
4         **do** INSERT$(U, r')$
5  **return** $U$
```

4.4 Data Structures

We have reduced the problem of implementing the region algebra to finding an efficient data structure for a rectangle collection that supports querying, insertion, and deletion. Research in computational geometry and multidimensional databases has resulted in a variety of suitable data structures. Samet [Sam90] gives a good survey.

This section describes three classes of spatial data structures that are useful for rectangle collections:

- *R-trees* divide the rectangle collection recursively. Each leaf of an R-tree is one rectangle in the collection, and each internal node stores the bounding box of the rectangles in its subtree. Variants of the R-tree variants use different insertion heuristics to minimize overlap between nodes, which speeds up query operations.

- *Quadtrees* divide region space recursively. Each quadtree node covers a fixed area of region space, and each node has four children, one for each equal quadrant of its area. Variants of the quadtree differ in how deeply the tree is subdivided and whether rectangles are stored only in the leaves or in all nodes.

- *Point-based methods* represent each rectangle as a four-dimensional point. The points are stored in a point data structure, such as a k-d tree, quadtree, or range tree.

Each class of data structure is described below. Discussion will focus on how to implement the three key operations for rectangle collections: querying for intersections with a rectangle, inserting a rectangle, and deleting a rectangle. The running time of these operations and the storage cost of the data structure will also be discussed.

Only one of these data structures is implemented in LAPIS: a variant of the R-tree that I call an *RB-tree*.

4.4.1 R-Trees

An *R-tree* [Gut84] is a balanced tree that stores an arbitrary collection of rectangles. The R-tree is based on the B-tree [CLR92], in that every internal node (other than the root) has between m and M children for some constants m and M, and the tree is kept in balance by splitting overflowing nodes and merging underflowing nodes. Each leaf node represents one rectangle in the collection, and each internal node stores the bounding box of all the rectangles in its subtree. An example R-tree is shown in Figure 4.7.

Querying for a rectangle in an R-tree uses Algorithm 4.6. The algorithm compares the query rectangle recursively against the bounding box of each node (*T.bbox)*. If the query rectangle does not intersect a node's bounding box, then the node's subtree is pruned from the search. When the search reaches a leaf of the tree, it returns the intersection of the query rectangle with the rectangle stored in the leaf (*T.rectangle*). For convenience, the pseudocode in Algorithm 4.6 uses the `yield` keyword from CLU [Lis81] to return each rectangle. Unlike `return`, `yield` implicitly saves a continuation so that the traversal can be resumed at the same point to generate the next rectangle.

Inserting a rectangle into an R-tree is similar to insertion into a B-tree. The new rectangle is inserted as a leaf. If the leaf's parent overflows (i.e., has $M + 1$ children), the parent is split into

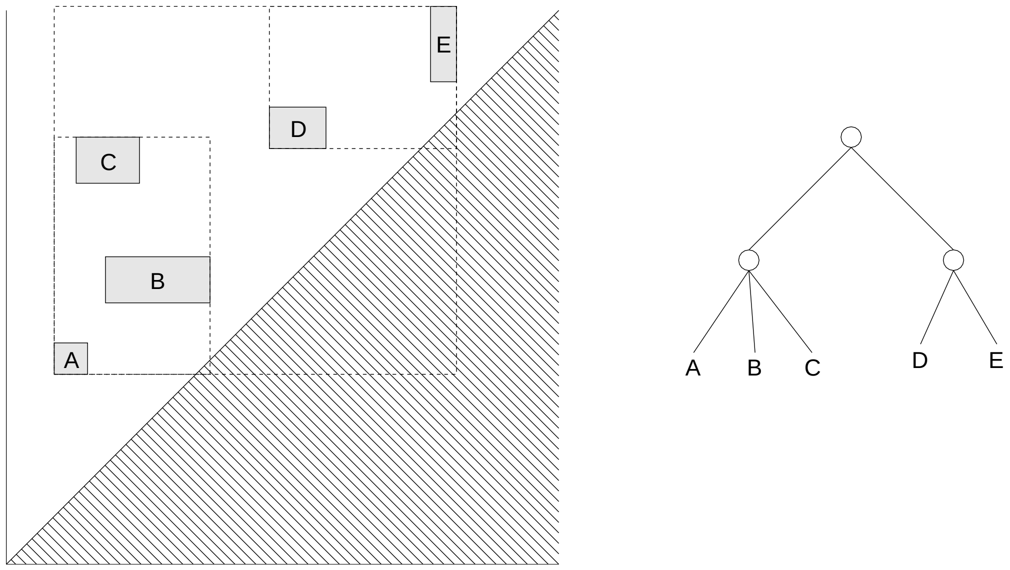


Figure 4.7: An R-tree containing 5 rectangles, $A - E$.

Algorithm 4.6 QUERY (T, r) traverses an R-tree T to find all rectangles that intersect the rectangle r.

```
QUERY$(T, r)$
1  **if** $r$ doesn't intersect $T.bbox$
2     **then return**
3  **if** $T$ is leaf
4     **then yield** $r \cap T.rectangle$
5     **else for each** $C$ **in** $T.children$
6              **do** QUERY$(C, r)$
```

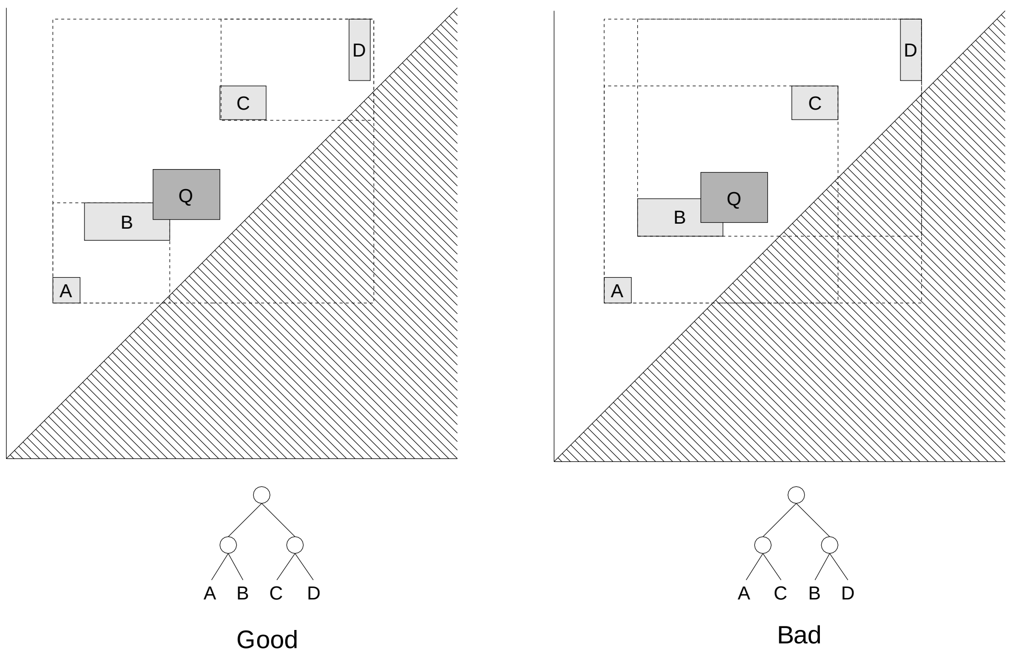


Figure 4.8: Different R-trees for the same set of rectangles can have different querying performance. A query for rectangle Q can avoid visiting the C-D subtree in the R-tree on the left, but it must visit all nodes in the R-tree on the right.

two nodes, which are then inserted into the grandparent. Overflows propagate up the tree as far as necessary. When the root overflows, it is split and a new root node is created, increasing the height of the tree.

Unlike a B-tree, however, an R-tree has no fixed rule for placing rectangles in its leaves. A rectangle can be inserted as *any* leaf without violating the R-tree's invariant properties. Bad placement leads to slower querying, however, because the internal nodes' bounding boxes become larger, more likely to overlap, and less likely to be pruned from the search. Figure 4.8 shows an example. Ideally, good placement should minimize both the area of the nodes and the overlap between sibling nodes.

Variants of the R-tree differ in the heuristics they use to achieve good placement. Two decisions are made heuristically. First is the *insertion heuristic*, which determines where to insert a new rectangle. Second is the *node-splitting heuristic*, which partitions the children of an overflowing node into two new nodes.

The original R-tree proposed by Guttman [Gut84] uses heuristics that minimize the area of bounding boxes. The insertion heuristic traverses the tree recursively, choosing the node whose bounding box would be expanded the least (in area) by the new rectangle. Ties are broken by choosing the node with the smallest area. For the node-splitting heuristic, Guttman offered three possibilities:

- an exponential heuristic that does an exhaustive search of the 2^M possible partitions, searching for the partition that produces the smallest nodes. This heuristic is generally impractical, but serves as a good baseline for measuring the performance of other heuristics.

- a quadratic heuristic that chooses two rectangles as seeds and greedily adds the remaining rectangles to the node whose bounding box needs the least enlargement. The seeds are chosen so that their bounding rectangle R maximizes $\mathit{area}(R) - \mathit{area}(\mathit{seed}_1) - \mathit{area}(\mathit{seed}_2)$. Implementing the second heuristic requires testing all M^2 possible pairs of seeds.

- a linear heuristic identical to the quadratic heuristic except that the chosen seeds are the rectangles with maximum separation (in some dimension), which can be computed in $O(M)$ time.

Guttman's experiments suggested that the linear heuristic was as good as the other two, but later experimenters [BKSS90] argue that the quadratic heuristic is superior to the linear heuristic in many cases.

The R*-tree [BKSS90] uses another set of heuristics. For non-leaf nodes, the R*-tree uses the same insertion heuristic as the R-tree, minimum area enlargement. For leaf nodes, however, the R*-tree insertion heuristic chooses the leaf node whose overlap with its siblings would be enlarged the least. The cost of computing each node's overlap with its siblings is $O(M)$, so this heuristic takes $O(M^2)$ time. For the node-splitting heuristic, the R*-tree sorts the rectangles separately along each axis, chooses one axis for splitting, and then splits the sorted list into two groups with minimum overlap. The sorting axis is chosen to minimize the perimeters of the two groups. This heuristic is $O(MlogM)$. The R*-tree heuristics were empirically shown to be efficient for random collections of rectangles [BKSS90].

LAPIS originally used the R*-tree heuristics. Rectangle collections used by the region algebra are *not* particularly random, however. They tend to be nonoverlapping and distributed linearly along some dimension of region space, such as the x-axis, y-axis, or $45°$ line. For such sets, a fixed lexicographic ordering of rectangles works just as well and avoids expensive placement calculations entirely. The revised R-tree data structure in LAPIS, which I call an *RB-tree*, orders its leaves lexicographically by (s_1, e_1, s_2, e_2). Each internal node in an RB-tree is augmented with a pointer to the lexicographically smallest leaf in its subtree, which allows the insertion heuristic to find the correct place for a new rectangle among the leaves. (This is equivalent to the way a conventional B-tree intersperses keys with child pointers in internal nodes.) The node-splitting heuristic simply divides the children in half, preserving their order.

A rectangle r is deleted from an RB-tree by querying the tree for r. Every rectangle that intersects r is either deleted from the tree outright using the B-tree deletion algorithm, or else split into one or more new rectangles which are reinserted into the tree. The cases for each kind of overlap are shown in Figure 4.9.

The time to query an RB-tree in which N rectangles have been inserted is $O(N)$ in the worst case, because the query rectangle may intersect the bounding box of every internal node, even if it does not intersect any of the leaves. Inserting a rectangle in the tree takes $O(\log N)$ time. Deleting a rectangle must query the tree, so it takes $O(N)$ time in the worst case. The average case is better, as the performance measurements in Section 4.7 show; querying, insertion, and deletion are typically $O(\log N)$.

The storage required by an RB-tree containing N rectangles is $O(N)$.

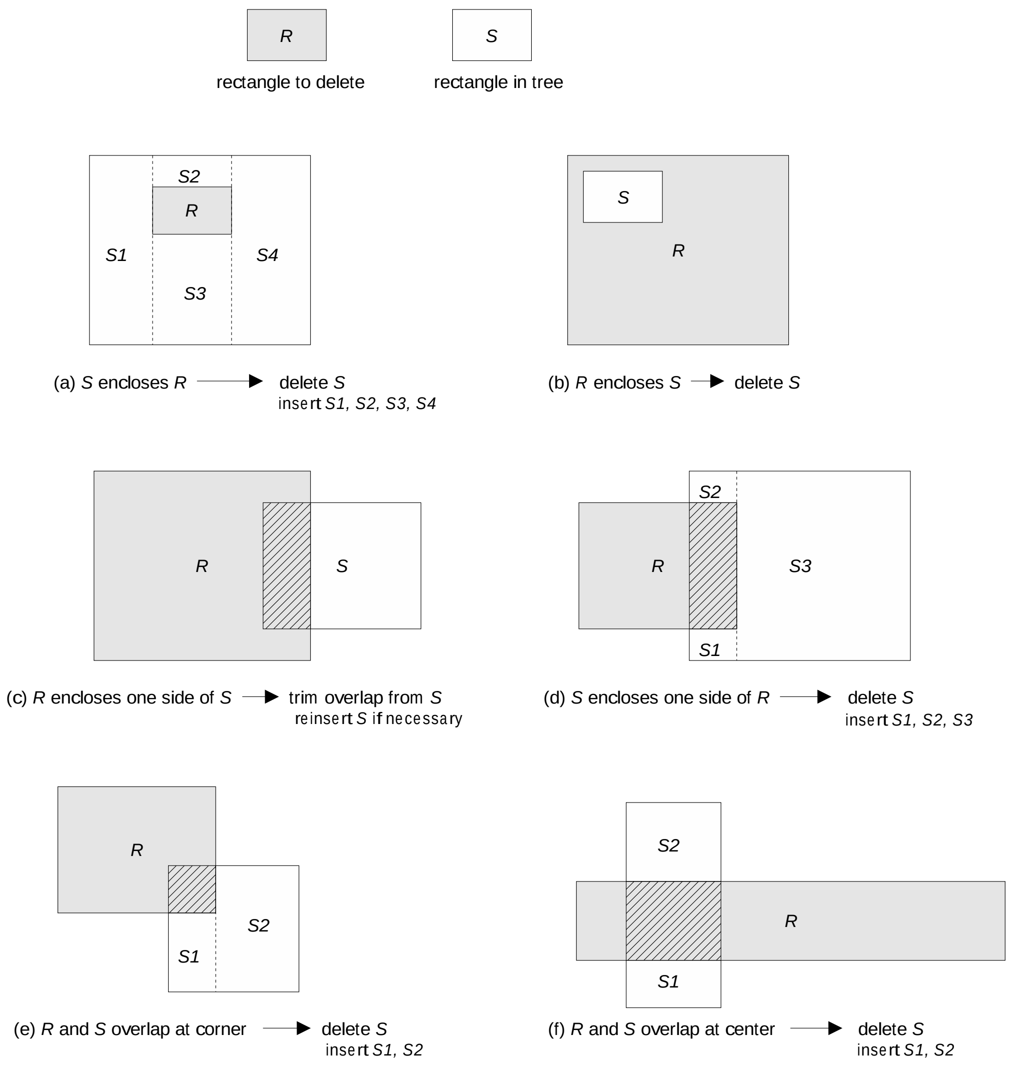

R
rectangle to delete
S
rectangle in tree
S2
R
S1
S4
S3
(a) S encloses R → delete S
insert S1, S2, S3, S4
S
R
(b) R encloses S → delete S
R
S
(c) R encloses one side of S → trim overlap from S
reinsert S if necessary
S2
R
S3
S1
(d) S encloses one side of R → delete S
insert S1, S2, S3
R
S2
S1
(e) R and S overlap at corner → delete S
insert S1, S2
S2
R
S1
(f) R and S overlap at center → delete S
insert S1, S2

Figure 4.9: Deleting a rectangle from an RB-tree.

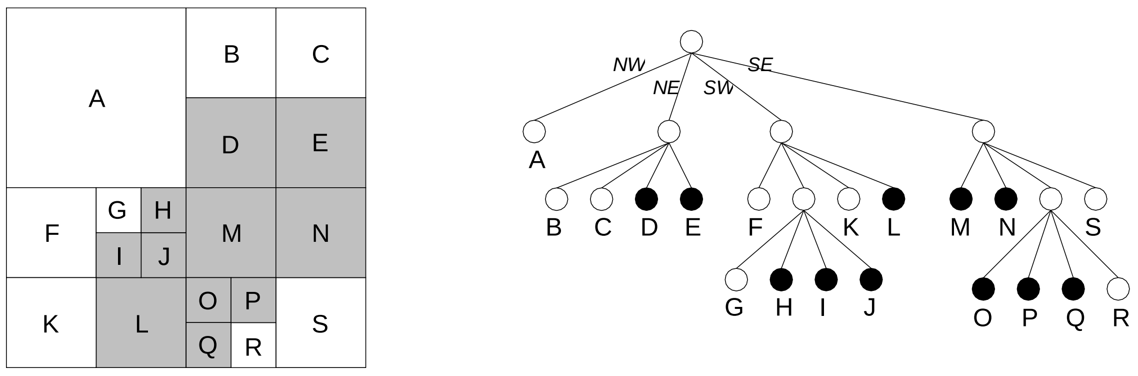


Figure 4.10: Example of a region quadtree (after Figure 1.1 from Samet [Sam90]).

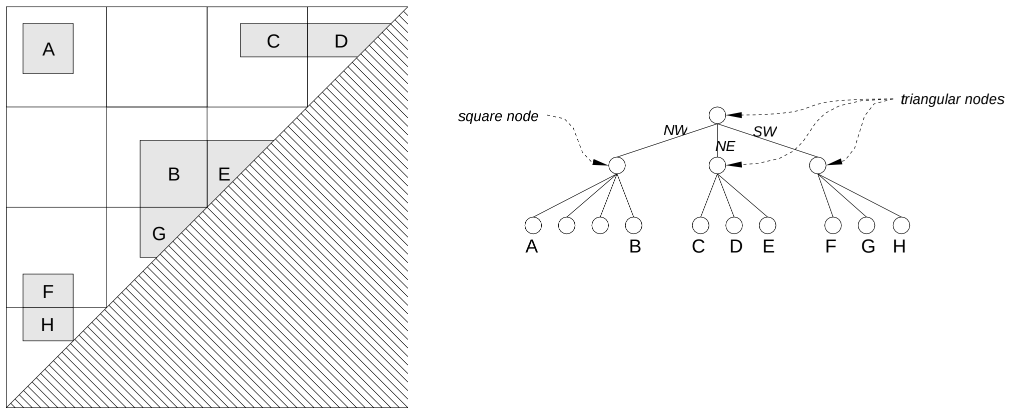


Figure 4.11: A quadtree used as a rectangle collection.

4.4.2 Quadtrees

A *quadtree* is a recursive data structure over 2D space. Each quadtree node represents a fixed area of space. The root node's rectangle is the entire space. Every node has four children, which divide the node's rectangle into four equal quadrants.

The most familiar kind of quadtree is the *region quadtree*, which is used to represent a set of points in the plane, such as a shape or a set of pixels in a raster image. A region quadtree is subdivided until its leaves are all homogeneous — either all points covered by the leaf are included in the set, or all points are excluded. A single bit in each leaf indicates which is the case. An example of a region quadtree is shown in Figure 4.10.

A region quadtree can store a rectangle collection by storing the set of points in the union of the inserted rectangles. The leaves need not be completely homogeneous, however. It is enough to subdivide until every leaf contains a rectangular set of points (or no points at all). Furthermore, region space is actually *triangular*, not square. As a result, quadtree nodes that intersect the $45°$ line only need three children, not four. These optimizations produce quadtrees like the one shown in Figure 4.11.

Querying a quadtree for a rectangle uses the same algorithm as the R-tree (Algorithm 4.6). One important difference is that a quadtree does not need to store node bounding boxes explicitly, because every node's bounding box is uniquely determined by its path from the root. Recursive algorithms like QUERY calculate bounding boxes on the fly as the quadtree is traversed.

Inserting a rectangle into a quadtree uses a similar recursive search for all nodes that intersect the new rectangle (Algorithm 4.7). If the new rectangle completely covers a quadtree node, then the node is simply converted to a leaf. Otherwise, if the node is a leaf, INSERT first tries to extend the leaf's rectangle to include the new rectangle. If the resulting shape would be nonrectangular, the leaf is split using SPLIT (Algorithm 4.8), and the new rectangle is inserted into its children. Node splitting is guaranteed to terminate, because eventually it will reach a 1×1 node which is either inside or outside the new rectangle. After inserting the new rectangle in the children of a node, the algorithm calls MERGE (Algorithm 4.9) to test whether the node now represents a simple rectangular area, in which case its children can be discarded and the node converted into a leaf. SPLIT and MERGE grow and shrink the quadtree to preserve the invariant that nodes are subdivided only as far as necessary to make every leaf contain a simple rectangular region.

Algorithm 4.7 INSERT (T, r) inserts a rectangle r into a quadtree T.

```
INSERT($T, r$)
1   if $r$ doesn't intersect $T.bbox$
2      then return
3   $r' \leftarrow r \cap T.bbox$
4   if $r' = T.bbox$
5      then delete children of $T$
6           $T.rectangle \leftarrow r'$
7           return
8   if $T$ is leaf
9      then if $T.rectangle \cup r'$ is rectangular
10              then $T.rectangle \leftarrow T.rectangle \cup r'$
11                   return
12              else SPLIT($T$)
13  for each $C$ in $T.children$
14  do INSERT($C, r$)
15  MERGE($T$)
```

Algorithm 4.8 SPLIT (T) converts a quadtree leaf into a node with children.

```
SPLIT($T$)
1  create children of $T$
2  INSERT($T, T.rectangle$)
3  $T.rectangle \leftarrow \emptyset$
```

Deleting a rectangle from a quadtree follows the same pattern as insertion. The only difference is that the query rectangle is subtracted from the tree's leaf rectangles, instead of added.

The running times of these algorithms depend on the height of the tree, which in turn depends on the size of region space. In a string of length n, region space has dimension $n \times n$, and any

Algorithm 4.9 MERGE (T) converts a quadtree node into a leaf if its children are all leaves and the union of their rectangles is rectangular.

```
MERGE($T$)
1  $r \leftarrow \emptyset$
2  for each $C$ in $T.children$
3  do if $C$ is not a leaf or $r \cup C.rectangle$ is not rectangular
4        then return
5     $r \leftarrow r \cup C.rectangle$
6  delete all children of $T$
7  $T.rectangle \leftarrow r$
```

quadtree over region space has $O(\log n)$ height. Querying or deleting a rectangle from the quadtree takes $O(F \log n)$ time, where F is the number of leaves in the tree that intersect the query. Inserting a rectangle takes $O(n)$ time in the worst case, because the quadtree may need to drill down to its maximum depth ($\log n$) in order to separate two rectangles into different nodes, and drilling down to maximum depth in one dimension can create up to n nodes in the other dimension (Figure 4.12). The storage required by a quadtree is $O(\min(nN, n^2))$ in the worst case.

This dependency on the size of region space n, rather than just the number of inserted rectangles N, is an important difference between quadtrees and RB-trees. The bounding box of each quadtree node is fixed, so a quadtree may be less efficient than an RB-tree at storing a collection of regions localized to a small part of a long string. On the other hand, the quadtree can completely eliminate overlaps between rectangles. Quadtrees are not implemented in LAPIS, so comparing the practical performance of RB-trees and quadtrees is left for future work.

The region quadtree is not the only kind of quadtree that could be used to implement a rectangle collection. For example, the MX-CIF quadtree [Sam90] associates each rectangle with the smallest quadtree node that completely contains it. This raises the question of how to organize the possibly-large list of rectangles stored on each node. One approach is an unordered list, but a more interesting approach reduces the dimension of the problem by 1: each rectangle is intersected with the node's horizontal and vertical axis, producing two sets of intervals that are stored in a pair of 1-dimensional MX-CIF quadtrees. Another quadtree is the RR quadtree [Sam90], which comes in two variants. The RR_1 quadtree splits nodes until each leaf intersects just one rectangle or a *clique*, a set of rectangles that are all mutually intersecting. The RR_2 quadtree splits until each leaf intersects a single rectangle or a chain of intersecting rectangles. Unlike the region quadtree, both MX-CIF and RR quadtrees guarantee to preserve the identities of the inserted rectangles — a guarantee which is not important for our rectangle collections.

4.4.3 Rectangles as Points

Another way to look at a rectangle (s_1, e_1, s_2, e_2) is as a point in four-dimensional space. This is the same change in perspective that led from regions as intervals in a one-dimensional string to points in two-dimensional region space (Section 3.3).

Representing rectangles as points makes it possible to store a rectangle collection in a data structure designed for points. In order to implement the QUERY operation, the point data structure

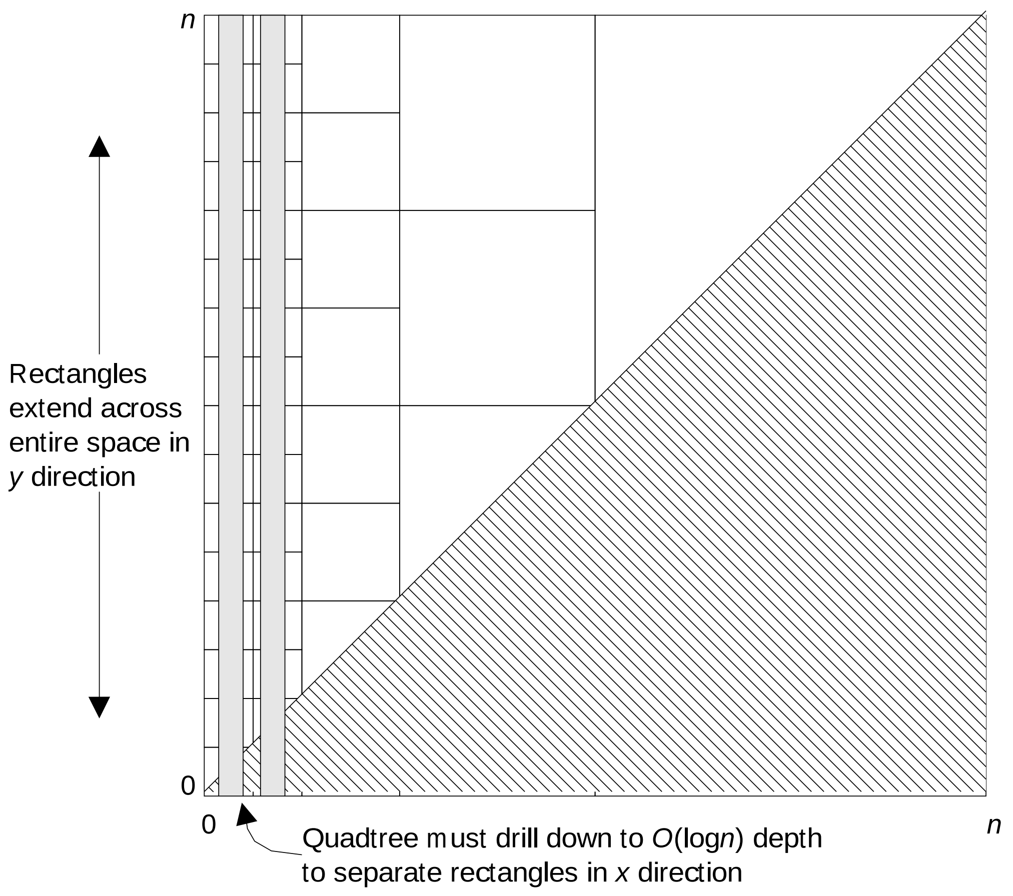

n
0
0
n
Rectangles extend across entire space in y direction
Quadtree must drill down to O(logn) depth to separate rectangles in x direction

Figure 4.12: Bad case for a quadtree: two long but closely-spaced rectangles force the quadtree to drill down to depth $O(\log n)$, requiring $O(n)$ quadtree nodes.

must support *range queries*. A range query returns all points that lie in a four-dimensional hyper-rectangle — the Cartesian product of four intervals, one for each dimension. For example, to find all the 2D rectangles (4D points) that are enclosed by the 2D query rectangle (s_1, e_1, s_2, e_2), one would use the range query $[s_1, s_2] \times [e_1, e_2] \times [s_1 : s_2] \times [e_1 : e_2]$. To implement QUERY, which searches for all the rectangles that intersect the query rectangle, one would use the range query $[-\infty, s_2] \times [-\infty, e_2] \times [s_1, \infty] \times [e_1, \infty]$.

A menagerie of data structures have been developed for k-dimensional points with range queries. Here are a few:

- The *k-d tree* [Ben75] is a binary tree in which each node stores a point, and each level of the tree compares a different dimension of a query point with its stored point. For example, in a 2-d tree, nodes at even-numbered depths might compare the x coordinates of the query point and the stored point, while nodes at odd-numbered depths compare the y coordinate. A new point is inserted by traversing the tree for the new point's correct position and adding it as a leaf. Like binary search trees, the performance of a k-d tree is very sensitive to the order in which the points are presented. An optimal k-d tree can be built in $O(kN \log N)$ time if all N points are known in advance. Range queries in an optimal k-d tree take $O(kN^{1-1/k})$ time in the worst case [LW77]. Since $k = 4$ for the purpose of this section, this means that querying takes $O(N^{3/4})$ worst-case time.

- A quadtree can also be used to store points. The 2D quadtree described in the previous section is generalized to k dimensions by splitting each node into 2^k children, one for each hyperquadrant. Many nodes will not need all 16 children, however, since some parts of 4-dimensional space can never contain a 4D point representing a 2D rectangle.

- The *point quadtree[FB74]* is a kind of quadtree that provides an adaptive decomposition of space. Each node stores one point from the set, and the node is split into 2^k children by hyperplanes passing through the stored point (instead of the node's centroid, as in the standard quadtree). A point quadtree is like a k-d tree in which each level discriminates on all k dimensions at once, instead of just one dimension at a time. A variant of the point quadtree chooses an arbitrary point to partition each node, not necessarily a point in the collection. Points in the collection are stored only in the leaves, which makes them easier to delete. Range queries in point quadtrees take $O(kN^{1-1/k})$ worst-case time, just like k-d trees [LW77].

- The *range tree [BM80]* is a data structure designed for good worst-case query performance at the cost of more storage. A one-dimensional range tree is a balanced binary search tree with the points stored in leaf nodes. The leaves are linked in sorted order by a doubly-linked list. A range query $[l, h]$ is done by searching the range tree for a leaf $\geq l$, then following the linked list until reaching a leaf $\geq h$. A k-d range tree uses a 1D range tree to index the x coordinate, and every node of the tree points to a $(k-1)$-d range tree indexing the remaining coordinates of every point in the node's subtree. The resulting data structure uses $O(N \log^{k-1} N)$ storage, and range queries can be satisfied in $O(\log^k N)$ time.

Although the point data structures have provably better asymptotic behavior than R-trees and quadtrees, using 4D makes the data structures more complicated and significantly increases the constant factors. Furthermore, points representing region rectangles are not uniformly distributed in 4D space. In many rectangle collections generated by the region algebra, the rectangles all share one or more coordinates, making those coordinates useless for discrimination — but point data structures give all coordinates equal weight. For these reasons, one would expect 4D point data structures to perform worse than R-trees or quadtrees on the average. No 4D data structures are implemented in LAPIS, so this comparison is left for future work.

As a special case, the 2D versions of these point data structures could be used to store rectangle collections where every rectangle is just a point. Flat, overlapping, and nested region sets all satisfy this requirement.

4.5 Specialized Data Structures

The rectangle-collection data structures that have been presented to this point — RB-trees, quadtrees, and 4D point collections — are suitable for storing arbitrary rectangle collections. For some important kinds of rectangle collections, however, there are simpler data structures.

- A *region array* is an array of rectangles in lexicographic order. Region arrays can store a *monotonic* rectangle collection, in which all rectangle coordinates increase monotonically. Flat and overlapping region sets are monotonic. Region arrays have guaranteed $O(\log N+F)$ query time.

- A *syntax tree* extends the familiar abstract syntax tree by storing a rectangular bounding box on each node. Syntax trees can be used to store nested region sets.

These data structures are described in more detail in the next two sections.

4.5.1 Region Arrays

For some rectangle collections, an RB-tree has too much overhead. In many cases, we can throw away the internal nodes of the RB-tree, leaving only the leaves, a list of rectangles sorted in lexicographic order. I call the resulting data structure a *region array*. Eliminating the internal nodes saves space, but only a constant factor since leaves always outnumber internal nodes. More importantly, however, it can be shown that a query on a region array always takes $O(\log N + F)$ time, where F is the number of intersecting rectangles found, which is an improvement over the $O(N)$ worst-case bound for RB-trees.

A region array can be used whenever the rectangle collection satisfies the following property. A rectangle collection is *monotonic* if, when the collection is sorted in increasing lexicographic order, the coordinates of the rectangles are also sorted in increasing order. In other words, if r and r' are a pair of rectangles in the collection such that $r \leq r'$ in lexicographic order, then $r.s_1 \leq r'.s_1$, $r.s_2 \leq r'.s_2$, $r.e_1 \leq r'.e_1$, and $r.e_2 \leq r'.e_2$. Many rectangle collections generated by the region algebra are monotonic. In particular, all flat and overlapping region sets are monotonic. The rectangle collection produced by applying a unary relational operator to a flat or overlapping region set is also monotonic. See Figures 4.4–4.6 for examples of these kinds of rectangle collections.

A nested region set is *not* monotonic in general, however. Figure 4.4 includes a nested set which is nonmonotonic. Region arrays cannot be used to store general nested region sets.

Monotonicity makes it easy to find the bounding box for any contiguous range of a region array. Suppose a region array A contains N monotonic rectangles in lexicographic order, so $A[1] \leq A[2] \leq \cdots \leq A[N]$. Because the collection is monotonic, we know that $A[1]$ has the minimum s and e values of all the rectangles, and $A[N]$ has the maximum values, so the bounding box of the entire collection is $(A[1].s_1, A[1].e_1, A[N].s_2, A[N].e_2)$. In general, for any $i \leq j$, the bounding box of $A[i \ldots j]$ is $(A[i].s_1, A[i].e_1, A[j].s_2, A[j].e_2)$.

We can use this fact to query a region array as if it were a binary RB-tree whose internal nodes are constructed on the fly. Algorithm 4.10 demonstrates this approach. The algorithm is very similar to the query algorithm for RB-trees (Algorithm 4.6), but bounding boxes and children are calculated on the fly. This algorithm uses $O(\log N)$ space to traverse the region array recursively, however, and it isn't clear how many bounding boxes may unnecessarily intersect the query rectangle.

Algorithm 4.10 QUERY (A, i, j, r) searches a segment of a region array, $A[i...j]$, for all rectangles that intersect the rectangle r.

```
QUERY($A, i, j, r$)
1  $bbox \leftarrow (A[i].s_1, A[i].e_1, A[j].s_2, A[j].e_2)$
2   if $r$ doesn't intersect $bbox$
3      then return
4   if $i = j$
5      then yield $r \cap bbox$
6      else $m \leftarrow \lfloor (i+j)/2 \rfloor$
7            QUERY($A, i, m, r$)
8            QUERY($A, m+1, j, r$)
```

A better algorithm uses a separate binary search on each coordinate. Recall from Section 4.4.3 that a rectangle intersection query is equivalent to the range query $[-\infty, r.s_2] \times [-\infty, r.e_2] \times [r.s_1, \infty] \times [r.e_1, \infty]$. In a region array, each of these intervals corresponds to a bound on the array indexes of intersecting rectangles. For example, consider the first interval, which stipulates that the s_1 coordinate of an intersecting rectangle must be less than or equal to $r.s_2$. Using a binary search on the s_1 coordinates of the region array (which are guaranteed to be in order because the rectangles are monotonic), find the largest index k_1 such that $A[k_1].s_1 \leq r.s_2$. Then $A[1 \ldots k_1]$ is the set of rectangles that satisfy the first interval. Similar binary searches for the other dimensions find $A[1 \ldots k_2]$ satisfying the second interval, $A[k_3 \ldots N]$ satisfying the second interval, and $A[k_4 \ldots N]$ satisfying the fourth interval. Putting all four constraints together, we find that the set of intersecting rectangles is precisely the rectangles in $A[\max(k_3, k_4) \ldots \min(k_1, k_2)]$. Algorithm 4.11 restates this procedure in pseudocode. This algorithm takes $O(\log N + F)$ time and $O(1)$ working space in all cases.

Region arrays are not designed for dynamic construction, so they do not support the INSERT operation. To construct a region array, LAPIS accumulates the rectangles resulting from an algebra operation. After the entire collection has been generated, the rectangles are sorted, and then a single pass over the collection tests whether the collection is monotonic. If so, the sorted list can be used directly as a region array. If not, the region array can be converted to an RB-tree by using the sorted list as a foundation and erecting the tree above it in $O(N)$ time.

Algorithm 4.11 QUERY (A, r) searches a region array for all rectangles that intersect the rectangle r, using a binary search on each coordinate.

```
QUERY$(A, r)$
1  $k_1 \leftarrow$ BINARYSEARCH$(A.s_1, r.s_2 + 1) - 1$
2  $k_2 \leftarrow$ BINARYSEARCH$(A.e_1, r.e_2 + 1) - 1$
3  $k_3 \leftarrow$ BINARYSEARCH$(A.s_2, r.s_1)$
4  $k_4 \leftarrow$ BINARYSEARCH$(A.e_2, r.e_1)$
5  for $i \leftarrow max(k_3, k_4)$ to $min(k_1, k_2)$
6  do yield $A[i]$

BINARYSEARCH$(L, x)$
1  do a binary search on array $L$ for value $x$
2  return the index $i$ such that $L[1 \ldots i-1] < x \leq L[i \ldots N]$
```

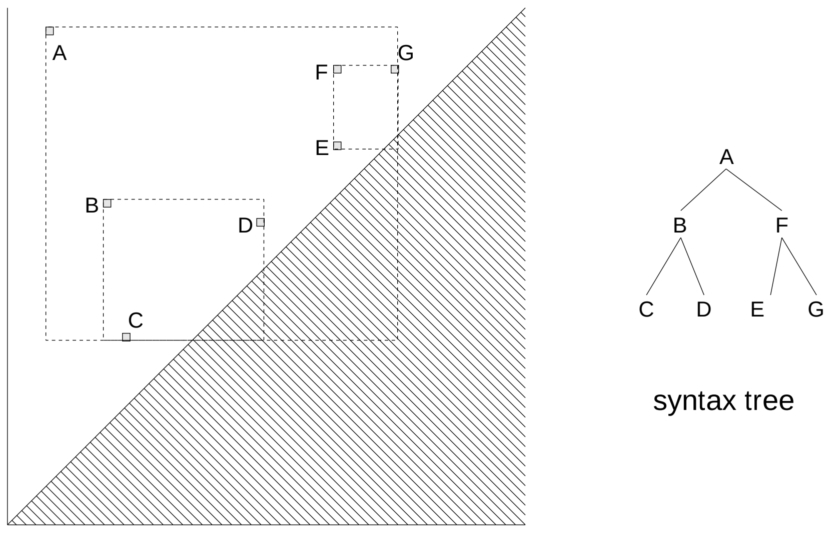
Deleting rectangles from a region array is possible only in certain cases. Deleting entire rectangles is easy, since a rectangle can be removed without affecting the monotonicity of the collection. A rectangle can be removed from the array in $O(1)$ amortized time by just marking it deleted. The array is reallocated and compacted only when at least half of its entries have been deleted. Harder cases arise when deletion would split or shrink a rectangle. Although it might be possible in some cases to split or shrink the rectangle in such a way that the collection is still monotonic (and doesn't require another $O(N \log N)$ sort), in general it is necessary to convert the region array to an RB-tree and then perform the deletion.

A region array of N rectangles takes $O(N)$ space. A region array saves up to a factor of 2 relative to an RB-tree by eliminating the internal nodes. LAPIS saves more space by representing the rectangles as an array of integers, eliminating the overhead of representing each rectangle as a Java object (12 bytes per object in Java 1.3). In LAPIS, a region array consumes about 16 bytes per rectangle, while an RB-tree averages 33 bytes per rectangle (counting the cost of internal nodes, with the tree's minimum and maximum branching factors set to 4 and 12, respectively). More space could be saved by delta-encoding or otherwise compressing the coordinates of the rectangles, at the cost of increased access time.

4.5.2 Syntax Trees

Context-free parsing often produces an abstract syntax tree representing the parse. If this syntax tree is augmented with the region from which each node was parsed, then the tree can be queried as if it were a rectangle collection.

In general, a *syntax tree* can represent any nested region set, not just the output of a context-free parser. In a syntax tree, any node may store a region, not just the leaves. When a region r is associated with a node, then the node's subtree must contain all the regions in the set that are *in* r. Nodes may have an arbitrary number of children, sorted in lexicographic order. Some nodes may have no associated region; such nodes are *empty*. Empty nodes arise when regions are deleted from the set. Empty nodes may also be used to ensure that the tree has a constant branching factor. Every node, empty or not, stores the bounding box of all the regions in its subtree.



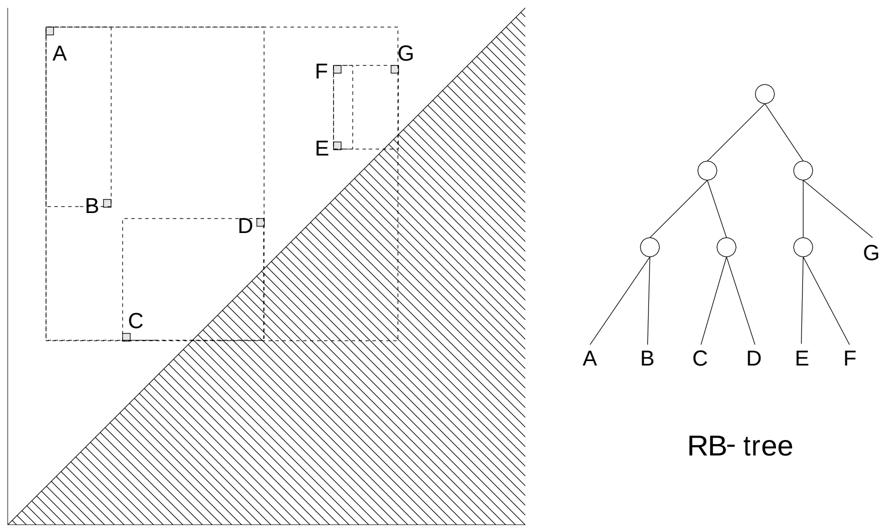


Figure 4.13: Syntax tree and RB-tree representations of a nested region set.

Figure 4.13 shows a syntax tree and an RB-tree for the same nested set. One key difference is that the syntax tree can store a region in any node, while the RB-tree stores regions only in leaves. Another difference is the shape of the node bounding boxes. RB-trees pack regions into the leaves in lexicographic order, without much concern for whether the regions are near each other in region space. Syntax trees can sometimes achieve more locality by exploiting the nested nature of the region set.

A syntax tree is queried by Algorithm 4.12. Every node in the syntax tree has two attributes: $T.bbox$ is the bounding box of the regions in T's subtree, and $T.region$ is the region stored in T itself. Since the bounding boxes of a node's children form a monotonic rectangle collection, the exhaustive search in line 5 can be replaced by a binary search as in region arrays, but this would help only when nodes have many children.

Algorithm 4.12 QUERY (T, r) searches a syntax tree for all regions that intersect the rectangle r.

```
QUERY($T, r$)
1  if $r$ doesn't intersect $T.bbox$
2     then return
3  if $r$ intersects $T.r$
4     then yield $r \cap T.r$
5  for each $C$ in $T.children$
6  do QUERY($C, r$)
```

A new region r can be inserted in a syntax tree by drilling down through nodes that *contain* r until finding a node whose children's bounding boxes are all *before* or *after* r, then inserting r in sorted order among the children. If some child's bounding box *overlaps-start* or *overlaps-end* r, then the resulting region set will no longer be nested, so the syntax tree must be converted to an RB-tree. The conversion is done by traversing the syntax tree in lexicographic order (Algorithm 4.13), from which an RB-tree can be built in $O(N)$ time.

Algorithm 4.13 LEXORDER (T) generates the regions from a syntax tree in lexicographic order. The algorithm is basically a preorder traversal, except that all descendants of T that start at the same point as T must be returned before T itself.

```
LEXORDER($T$)
1  PRE($T$)
2  POST($T$)

PRE($T$)
1  if $T$ is not a leaf and $T.children[1].bbox.s_1 = T.bbox.s_1$
2     then PRE($T.children[1]$)
3  if $T.region \neq \emptyset$
4     then yield $T.region$

POST($T$)
1  if $T$ is not a leaf and $T.children[1].bbox.s_1 = T.bbox.s_1$
2     then POST($T.children[1]$)
3  for each $C$ in $T.children$ such that $C.bbox.s_1 > T.bbox.s_1$
4  do LEXORDER($C$)
5
```

A region can be deleted from a syntax tree by removing it from its node (i.e., setting $T.region$ to $\emptyset$) and updating its ancestors' bounding boxes. If a subtree becomes completely empty, with no regions associated with any of its nodes, it can be pruned from the tree.

Syntax trees are not currently implemented in LAPIS.

4.6 Optimizations

The basic region algebra implementation should now be clear. A region set is stored in a rectangle collection data structure, such as an RB-tree or a region array. Algebra operators combine the region sets by applying relational operators to rectangles, intersecting rectangle collections, merging rectangle collections, or deleting rectangles from a rectangle collection.

The basic implementation can be tweaked in many ways to improve its performance. The following optimizations are discussed in this section:

- Trimming mutually-intersecting rectangles in a rectangle collection.
- Generating rectangle collections in lexicographic order wherever possible to avoid sorting.
- Doing set operations on collections of single-point rectangles using a sorted merge in $O(N+M)$ time.
- Intersecting rectangle collections by traversing both trees in tandem.
- Doing set operations on quadtrees by traversing both trees in tandem.
- Intersecting rectangle collections using the optimal plane-sweep algorithm.

Some of these optimizations are implemented in LAPIS, but others are left for future work.

4.6.1 Trimming Overlaps

Query performance is dramatically reduced when many of the rectangles in a collection intersect each other. For example, as shown in Figure 4.5, the *before, after,* and *contains* operators produce rectangle collections in which all N rectangles are mutually intersecting. Querying one of these rectangle collections, even with a small query rectangle, may produce up to $O(N)$ rectangles as a result.

This problem can be addressed by trimming rectangles when they are inserted into the collection, in order to eliminate as much overlap as possible between the new rectangle and existing rectangles. Figure 4.14 illustrates the trimming heuristics:

1. If the new rectangle is completely enclosed by some rectangle already in the collection, then the new rectangle is not inserted (Figure 4.14(a)).
2. If some rectangle in the tree is completely enclosed by the new rectangle, then the enclosed rectangle is deleted from the tree (Figure 4.14(b)).
3. If some existing rectangle encloses one whole side of the new rectangle, then the overlapping part is subtracted from the new rectangle (Figure 4.14(c)).
4. If the new rectangle encloses a side of an existing rectangle, then the existing rectangle is trimmed and reinserted (Figure 4.14(d)).

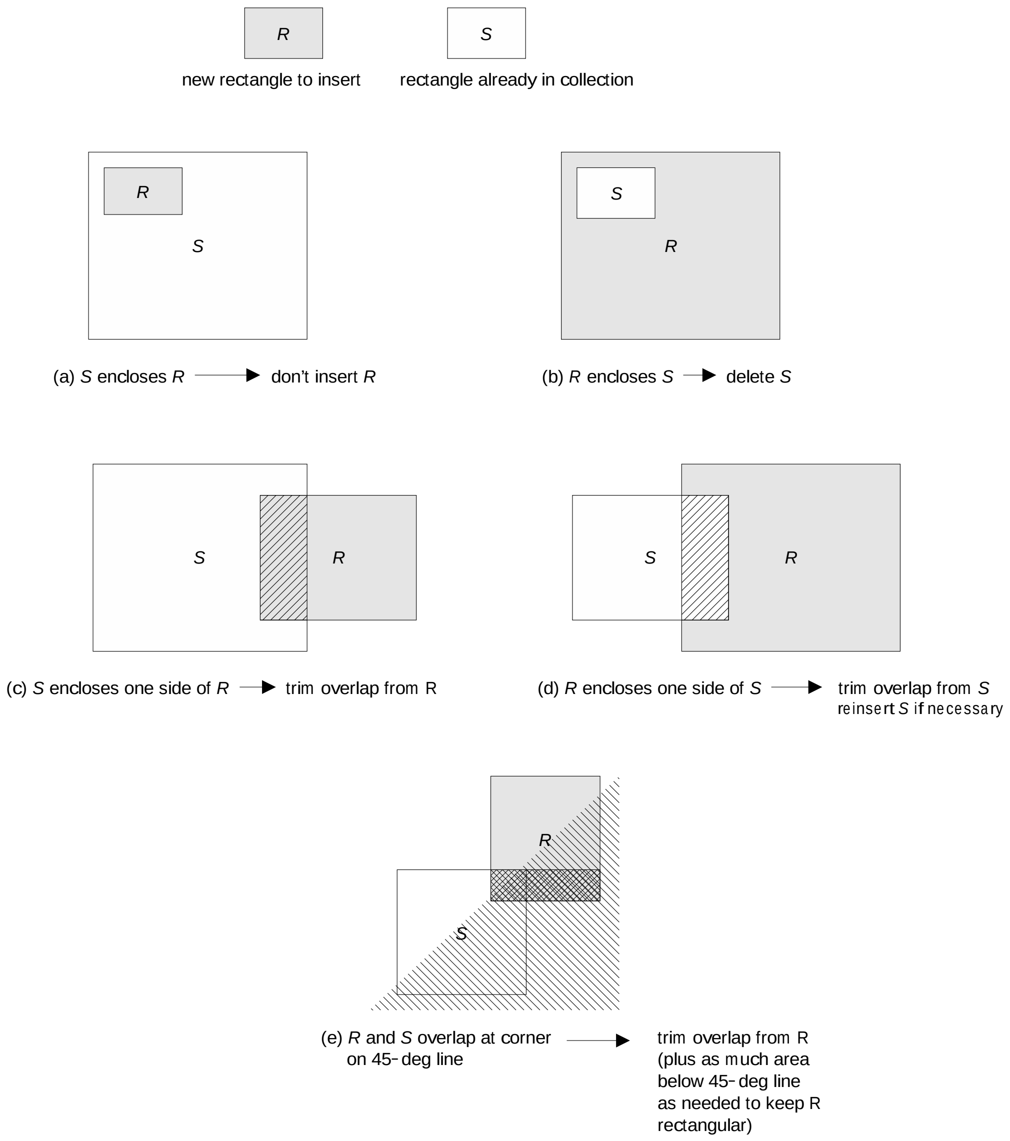

R
S
new rectangle to insert
rectangle already in collection
(a) S encloses R → don't insert R
(b) R encloses S → delete S
(c) S encloses one side of R → trim overlap from R
(d) R encloses one side of S → trim overlap from S reinsert S if necessary
(e) R and S overlap at corner on 45-deg line → trim overlap from R (plus as much area below 45-deg line as needed to keep R rectangular)

Figure 4.14: Heuristics for reducing overlap in a rectangle collection.

5. If the new rectangle and the old rectangle intersect at a corner which is cut by the $45°$ line of region space, then the overlapping part is subtracted from the new rectangle, along with enough area below the $45°$ line to keep it rectangular (Figure 4.14(e)).

Not all overlaps can be eliminated by these heuristics, since rectangles can still overlap at a corner away from the the $45°$ line, or through the center, but these heuristics help reduce overlap in the common cases produced by the region algebra.

Trimming all rectangles against each other might take $O(N^2)$ time in general, since each rectangle must be queried against the rest of the collection. LAPIS takes a simpler approach that works well for rectangle collections produced by the region algebra. After generating a list of rectangles and sorting them lexicographically, LAPIS makes a single pass through the sorted list, trimming each pair of rectangles. The effects of the heuristics on some common rectangle collections are shown in Figure 4.15.

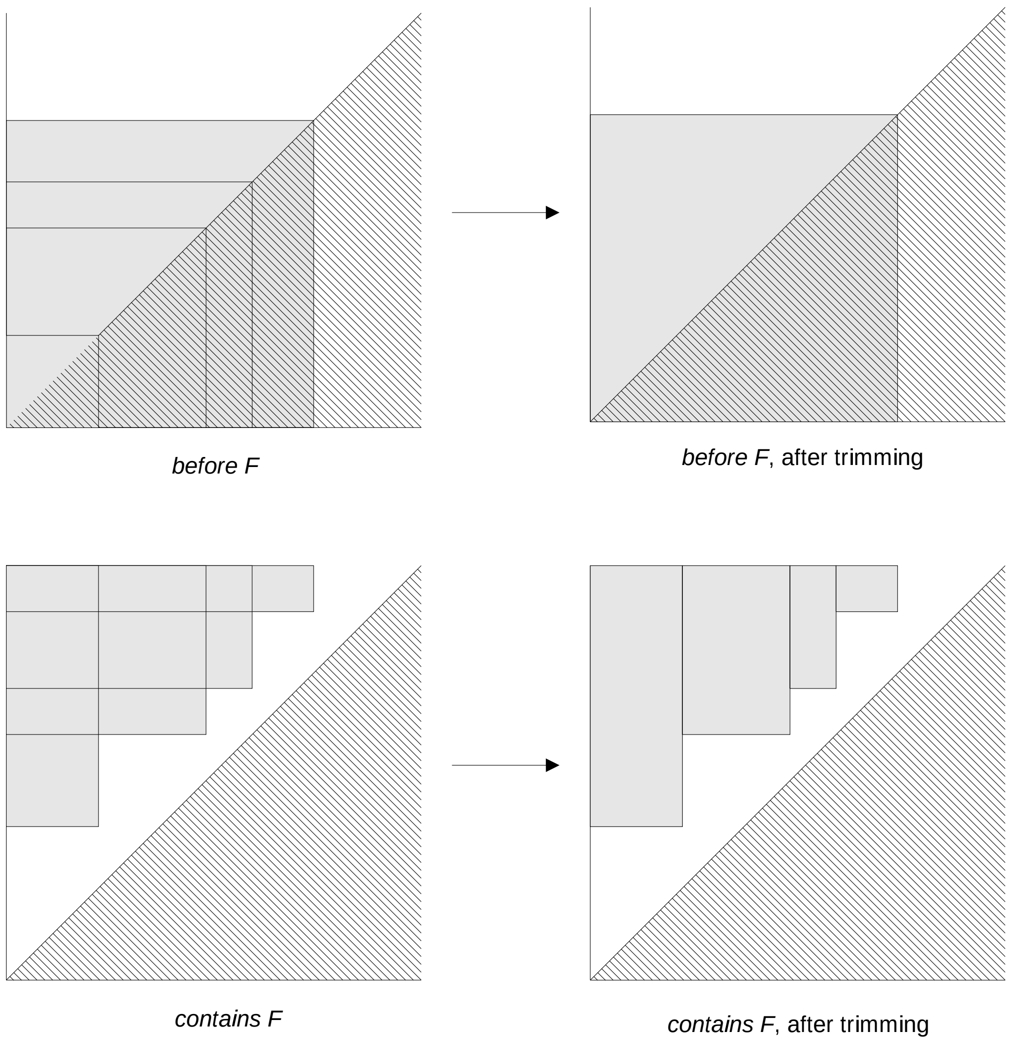


Figure 4.15: Rectangle collections after reducing overlap.

4.6.2 Preserving Lexicographic Order

Most of the cost of constructing an RB-tree or a region array is sorting the rectangle collection. If we can guarantee that the process generating the rectangles generates them in sorted order, then the sorting step can be omitted, and the RB-tree or region array can be built in $O(N)$ time.

Literal string matching and regular expression matching are two processes that naturally generate regions in lexicographic order. Each of these processes does a left-to-right scan over the string, generating all matches that start at position i before considering position $i + 1$.

Context-free parsing does *not* naturally generate regions in lexicographic order. For example, consider parsing the expression $a + b$ with conventional left-to-right, shift-reduce parsing. The parser first shifts a onto its stack. It then reduces a to an *Expression* nonterminal and emits a as a region for the *Expression* region set. Next, the parser shifts $+$ and b, reduces b to an expression and emits b as an *Expression* region, and then finally reduces *Expression*$+$*Expression* and emits $a + b$ as an *Expression* region. The resulting stream of regions — a, b, $a + b$ — is not in lexicographic order, because $a + b$ should precede b lexicographically.

Nevertheless, a context-free parser can produce a sorted stream of regions in $O(N)$ time if it first creates a syntax tree representing the parse. The tree is then scanned using Algorithm 4.13 to produce the region set in lexicographic order.

Applying a relational operator like *in*, *contains*, *starting*, or *ending* to a monotonic rectangle collection in lexicographic order always produces its result in lexicographic order. In fact, the result is itself a monotonic rectangle collection, as the following argument shows. Take any two rectangles $r < r'$ in the monotonic rectangle collection. Because of monotonicity, every coordinate of r is less than or equal to the corresponding coordinate of r'. Applying any relational operator *op* to r produces a new rectangle $\mathit{op}\, r$ that just rearranges the coordinates of r (possibly substituting 0 or n for some coordinates). This fact can be verified in Figure 4.3. Thus every coordinate of $\mathit{op}\, r$ is less than or equal to the corresponding coordinate of $\mathit{op}\, r'$, so $\mathit{op}\, r \leq \mathit{op}\, r'$ lexicographically and $\mathit{op}\, r$ and $\mathit{op}\, r'$ are monotonic. As a consequence, a relational operator can be applied to a region array simply by copying the region array and replacing each rectangle r with $\mathit{op}\, r$. Alternatively, the relational operator can be implemented lazily by a wrapper that applies *op* only when a rectangle is requested from the array. It takes only $O(1)$ time to apply the wrapper. LAPIS uses the lazy approach.

In general, the intersection, union, and difference operators do not necessarily generate their results in lexicographic order. One important case in which they do is when one or both operands is a collection of *one-point rectangles*. This case is described in the next section.

4.6.3 Point Collections

A *point collection* is a rectangle collection consisting entirely of one-point rectangles (s, e, s, e). Technically, any rectangle collection can be converted to a point collection by exploding its rectangles into individual points, but in general this would result in a quadratic explosion in the size of the collection. I will restrict the use of the term *point collection* to region sets that are naturally represented as single points. For example, nested, flat, and overlapping region sets are point collections. Unions of (a small number of) point collections are also naturally represented as point collections. The intersection or difference of a point collection and any other rectangle collection is always a point collection.

The intersection, union, or difference of two point collections can be found in linear time by traversing both collections simultaneously in lexicographic order. Algorithm 4.14 illustrates how intersection is done. Union and difference are similar. Not only does this algorithm take only $O(M+N)$ time, compared to $O(M \log N)$ average time for set operations on general rectangle collections, but the result is guaranteed to be in lexicographic order.

Algorithm 4.14 INTERSECTION(C_1, C_2) intersects two point collections by traversing them in lexicographic order.

```
INTERSECTION($C_1, C_2$)
 1  $U \leftarrow$ new RECTANGLECOLLECTION
 2  $p_1 \leftarrow$ first point in $C_1$
 3  $p_2 \leftarrow$ first point in $C_2$
 4  while $p_1 \neq \emptyset$ and $p_2 \neq \emptyset$
 5  do  if $p_1 = p_2$
 6        then INSERT$(U, p_1)$
 7             $p_1 \leftarrow$ next point in $C_1$
 8             $p_2 \leftarrow$ next point in $C_2$
 9      else if $p_1 < p_2$
10        then $p_1 \leftarrow$ next point in $C_1$
11        else $p_2 \leftarrow$ next point in $C_2$
12  return $U$
```

Set operations involving point collections are more predictable (in the worst case) than set operations on general rectangle collections. We can exploit this fact by transforming an algebra expression into an equivalent expression that applies intersection, union, or difference to point collections instead of arbitrary rectangle collections, as much as possible. For example, the expression

$$\textit{Line} \cap ((\textit{starts}\ \text{"From:"} \cup \textit{starts}\ \text{"Sender:"} \cup \textit{contains}\ \text{"cmu.edu"}) \tag{4.1}$$

refers to four point collections: *Line* and the three quoted literals. In this expression, the union operators combine arbitrary rectangle collections generated by the unary relational operators *starts* and *contains*. The equivalent expression

$$((\textit{Line} \cap \textit{starts}\ \text{"From:"}) \cup (\textit{Line} \cap \textit{starts}\ \text{"Sender"})) \cup (\textit{Line} \cap \textit{contains}\ \text{"cmu.edu"}) \tag{4.2}$$

ensures that every union involves a point collection. In general, if we denote an expression known to return a point collection by P and other expressions by E, the transformation is described by the following rules, applied repeatedly until none match:

$$\begin{aligned} P \cap (E_1 \cap E_2) &\rightarrow (P \cap E_1) \cap E_2 \\ P \cap (E_1 \cup E_2) &\rightarrow (P \cap E_1) \cup (P \cap E_2) \\ P - E &\rightarrow P - (P \cap E) \end{aligned}$$

This transformation does not necessarily improve performance, however. In the example given, suppose that *Line* is much larger than the other region sets (“From”, “Sender”, “cmu.edu”). Then expression 4.2, which queries the *Line* region set three times, may actually run slower than the expression 4.1, which combines the three literals before querying *Line*. LAPIS does not apply the transformation automatically. An expert user might use it, however, to optimize the performance of a pattern.

Other pattern-matching systems, such as Proximal Nodes [NBY95] and WebL [KM98], can perform set operations only on point collections. The pattern languages in these systems are constrained (primarily by the absence of unary relational operators) so that only patterns like 4.2 can be written.

4.6.4 Reordering Operands

For commutative operators like intersection and union, the implementation is free to change the order of operands. LAPIS uses this fact to optimize both $A \cap B$ and $A \cup B$. The intersection operator enumerates the rectangles in the smaller collection and queries each rectangle against the larger collection. The union operator uses COPY to duplicate the larger collection (which never takes more than linear time, and so can be faster than rebuilding it) and then inserts rectangles from the smaller collection into the larger.

Optimizing the intersection operator also optimizes the binary relational operators, since they are implemented using intersection. Consider the expression:

Sentence contains "Gettysburg"

In a typical document, *Sentence* would have far more matches than "Gettysburg", so it would be faster to enumerate the matches to "Gettysburg" and query them against the *Sentence* region set. Since the internal representation of this expression is

Sentence $\cap$ *contains* "Gettysburg"

this optimization happens automatically. The intersection operator observes that *contains* "Gettysburg" contains fewer rectangles than *Sentence*, so it queries *contains* "Gettysburg" against *Sentence*.

The same optimization works for the complementary expression

"Gettysburg" *in Sentence*

LAPIS still queries the smaller region set derived from "Gettysburg" against the larger region set derived from *Sentence*, but this time it returns "Gettysburg" regions instead of *Sentence* regions. Note that, since *Sentence* is a flat region set, *in Sentence* can be computed from *Sentence* in $O(1)$ time by placing a wrapper around it, as described in Section 4.6.2.

4.6.5 Tandem Traversal

The simple algorithm for $A \cap B$ (Algorithm 4.3) takes each rectangle in B and traverses A recursively to find the intersections. I call this algorithm *iterative intersection.* The iterative algorithm takes no advantage of the fact that B is also a tree, hopefully organizing its rectangles with some locality. *Tandem intersection* exploits this by traversing both A and B simultaneously.

The tandem intersection algorithm is shown in Algorithm 4.15. The key line is line 1, which tests whether the bounding boxes of the entire trees A and B intersect. If their bounding boxes do not intersect, then we know that none of the rectangles stored in A can possibly intersect the rectangles stored in B. Thus, a single comparison high in the tree may be able to eliminate many comparisons at the leaves.

If the bounding boxes of A and B intersect after all, then the algorithm recursively drills into either A or B. The algorithm drills into A only if A's bounding box encloses B's bounding box, or B has no children; otherwise it drills into B.

Algorithm 4.15 TANDEMINTERSECTION finds the intersection of two tree-shaped rectangle collections by traversing both trees at the same time.

```
TANDEM($A, B$)
1  if $A.bbox$ doesn't intersect $B.bbox$
2     then return
3  if $A$ and $B$ are leaves
4     then yield $A.rectangle \cap B.rectangle$
5  else if $A.bbox \supseteq B.bbox$ or $B$ is a leaf
6     then for each $C$ in $A.children$
7              do TANDEM($C, B$)
8     else for each $C$ in $B.children$
9              do TANDEM($A, C$)
```

Tandem intersection is not guaranteed to be faster than iterative intersection. In the worst case, tandem intersection may do more work, because it can compare nodes of A with any node of B, while iterative intersection only compares nodes of A with leaves of B. However, the extra work is bounded by at most a factor of 2, as the following argument shows. We will compare the number of recursive calls made by tandem intersection with the number of recursive calls made by iterative intersection. Recall that the iterative intersection of A and B calls QUERY(A, r) for all rectangles $r \in B$. The following lemma relates the number of TANDEM calls to the number of QUERY calls:

Lemma 1. *For any two tree nodes A and B, if* TANDEM(A, B) *is called during a tandem intersection of two rectangle collections, then for all rectangles $r \in B$,* QUERY(A, r) *is called in the iterative intersection of the same collections.*

Proof. By induction on the depth of recursion. For the top-level call to TANDEM(A, B), A is the root of the tree, so iterative intersection must also start by calling QUERY(A, r) for all rectangles $r \in B$. For the induction step, we assume the hypothesis true for a call to TANDEM at recursion depth k, then show that the hypothesis is true for any recursive call it makes at recursion depth $k + 1$. There are four cases, corresponding to the conditions tested by Algorithm 4.15:

- $A.bbox \cap B.bbox = \emptyset$. In this case, tandem intersection makes no recursive calls to TANDEM, so there is nothing to prove.

- $A.bbox \supseteq B.bbox$. In this case, tandem intersection recursively calls TANDEM(C, B) for every child C of A. From the induction hypothesis, we know that iterative intersection calls

Query(A, r) for every $r \in B$. Since $r \subseteq B.bbox \subseteq A.bbox$, each of these Query calls must call Query(C, r) for all children C of A as well, so the hypothesis is proved for depth $k + 1$.

- B is a leaf. Like the previous case, tandem intersection drills into A. But there is only one rectangle in B, namely B itself. Thus, by the induction hypothesis, iterative intersection calls Query(A, B). Since $A.bbox$ and $B.bbox$ intersect, Query(A, B) must recursively call Query(C, B) for all children C of A, so the hypothesis is proved for depth $k + 1$.

- Otherwise, tandem intersection drills into B, recursively calling Tandem(A, C) for every child C of B. Since the induction hypothesis implies that Query(A, r) is called for all $r \in B$, and C is a subtree of B, we trivially have Query(A, r) for all $r \in C$. Thus the hypothesis is proved for depth $k + 1$.

□

Using Lemma 1, an amortized analysis shows that tandem intersection makes no more than twice as many calls to Tandem as iterative intersection would make to Query. We will charge the cost of calling Tandem(A, B) (not including its recursive calls) to all the Query(A, r) calls made with the rectangles in the leaves of B. Lemma 1 guarantees that all these Query calls are made. The share of the cost assigned to Query(A, r) is proportional to the depth of r in the tree, so that if B has branching factor m at every node and r is stored at depth d, then r is charged $1/m^d$ of the cost. Since every leaf of B contains a rectangle, the sum of these costs is 1. Turning the analysis around to look at it from the perspective of a Query call, Query(A, r) may be charged for some part of a Tandem(A, B) call for each B on the path from r to the root. If the minimum branching factor of the tree is m, then the cost charged to Query(A, r) is at most

$$1 + \frac{1}{m} + \frac{1}{m^2} + \frac{1}{m^3} + \cdots < \frac{m}{m-1}$$

which is at most 2 if $m \geq 2$. Thus the sum of the costs charged to the Query calls is at most twice the number of Query calls. Since this total cost is the same as the number of Tandem calls, there can be at most twice as many Tandem calls as Query calls.

So even in the worst case, tandem intersection is at most a factor of two worse than iterative intersection. In the average case, however, tandem intersection takes only $O(N)$, as the performance measurements in Section 4.7 show. This is significantly better than the $N \log N$ average-case time for iterative intersection. LAPIS uses tandem intersection exclusively.

Tandem intersection can be applied to any tree-like rectangle collection, including RB-trees, quadtrees, region arrays, and syntax trees. Since not all leaves in a quadtree contain a rectangle, however, the amortized analysis does not work for quadtrees, so tandem intersection on a quadtree may be more than a factor of 2 worse than iterative intersection. The next section discusses a form of tandem traversal specialized to quadtrees.

Tandem intersection also works when A and B are different data structures. For example, an RB-tree can be tandem-intersected with a region array or a syntax tree.

4.6.6 Quadtree Traversal

Quadtrees are particularly amenable to tandem traversal, because every quadtree decomposes region space in exactly the same way. To intersect two quadtrees A and B whose bounding boxes are the same, it suffices to recursively intersect only the four pairs of corresponding children, $A.children[i]$ with $B.children[i]$, as i ranges from 1 to 4. Algorithm 4.16 shows how this is done. As long as A and B are not leaves, QUADTREEINTERSECT traverses them in tandem. Whenever one tree reaches a leaf, the algorithm copies the other tree and calls INTERSECTWITH to intersect the leaf's rectangle with all the nodes in the other tree. When both functions return from their recursive traversal, they call MERGE (Algorithm 4.9) to test whether the intersected children can be merged into one node containing a single rectangle. The time to run the overall algorithm is proportional to the number of nodes in the quadtrees, which is $O(\min(nN, n^2))$ in the worst case.

Algorithm 4.16 QUADTREEINTERSECT finds the intersection of two quadtrees by tandem traversal.

```
QUADTREEINTERSECT($A, B$)
 1  if $A.children = \emptyset$
 2     then $U \leftarrow$ COPY$B$
 3          INTERSECTWITH($U, A.rectangle$)
 4  else if $B.children = \emptyset$
 5     then $U \leftarrow$ COPY$A$
 6          INTERSECTWITH($U, B.rectangle$)
 7     else $U \leftarrow$ new QUADTREE
 8           for $i \leftarrow 1$ to $4$
 9           do $U.children[i] \leftarrow$ QUADTREEINTERSECT($A.children[i], B.children[i]$)
10          MERGE($U$) return $U$

INTERSECTWITH($A, r$)
 1  if $r \cap A.bbox \neq \emptyset$
 2     then if $A.children = \emptyset$
 3             then $A.rectangle \leftarrow r \cap A.rectangle$
 4             else for each $C$ in $A.children$
 5                    do INTERSECTWITH($C, r$)
 6                   MERGE($A$)
```

Similarly, tandem traversal can can be used for the union or difference of two quadtrees. These algorithms save time relative to the general union and difference algorithms, which used INSERT and DELETE, because the cost of traversing the tree to insert (or delete) a rectangle is amortized over all the rectangles to be processed.

4.6.7 Plane-Sweep Intersection

Finding the intersections in a collection of rectangles is a well-studied problem in computational geometry. Traditionally, the rectangle-intersection problem is formulated as follows: given a col-

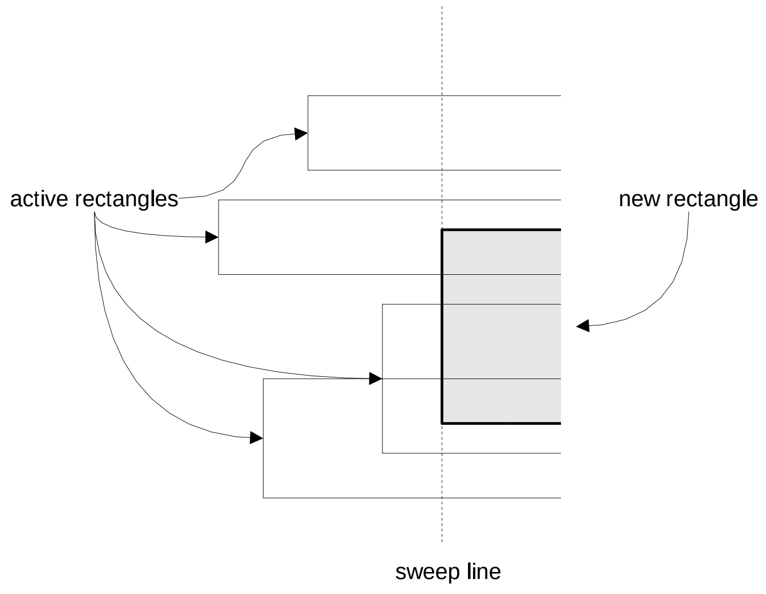


Figure 4.16: The plane-sweep intersection algorithm (after Preparata & Shamos [PS85], Figure 8.29).

lection of N axis-aligned rectangles, find all intersecting pairs of rectangles. The classic text by Preparata and Shamos [PS85] gives an optimal, $O(N \log N + F)$-time solution to this problem, where F is the number of intersections found. This section briefly outlines this algorithm, then describes how it can be used for region algebra intersection.

The algorithm uses the plane-sweep technique, passing a vertical sweep line through the rectangles from left to right (Figure 4.16). The sweep line stops at every x-coordinate of a rectangle, either its left side or its right side. The algorithm maintains a set of *active rectangles*, which are the rectangles currently intersecting the sweep line. When the left side of a rectangle is encountered by the sweep line, the rectangle is added to the active set. When its right side is encountered, it is deleted from the active set.

The active set is used to detect rectangle intersections. When a rectangle is added to the active set, the algorithm checks whether the new rectangle's y-interval intersects the y-interval of a currently-active rectangle. If so, then the rectangles are reported as an intersecting pair. In order to make this search fast, the active set's y-intervals are stored in an *interval tree*, a data structure that allows intervals to be inserted and deleted in $O(\log N)$ time, and handles range queries in $O(\log N + F)$ time. The interval tree was discovered independently by Edelsbrunner [Ede80] and McCreight [McC81]. See Preparata & Shamos [PS85] for more details.

To find the intersections in a collection of N rectangles, the plane sweep algorithm needs $O(N \log N)$ preprocessing time to separately sort the x-coordinates of the rectangles for the plane sweep and the y-coordinates for initializing the interval tree. The plane sweep itself takes $O(N \log N + F)$ time, so the overall time is $O(N \log N + F)$. Preparata and Shamos prove that this time is optimal for a decision-tree algorithm (i.e., one that only makes comparisons between rectangle coordinates).

For the region algebra, we want to solve a slightly different problem: given *two* collections of rectangles A and B, find all intersecting pairs (a, b) such that $a \in A$ and $b \in B$. One solution is to combine both collections, $A \cup B$, marking each rectangle according to whether it came from A or B (or both). Then find the intersecting pairs in the union and discard all but the (a, b) pairs. The problem with this approach is that finding and reporting the self-intersections, (a, a') and (b, b'), may dominate the running time of the algorithm, making it $O(N^2)$ in the worst case.

Instead, we maintain two active sets, one for A and one for B. The sweep line passes across the union of both collections. When an A rectangle becomes active, its y-interval is tested against B's active set to find intersecting pairs, but then inserted into A's active set. Vice versa for B. If the two collections have size M and N respectively, then this algorithm takes $O((M+N)\log(M+N))$ time to sort the x-coordinates of both collections together for the plane sweep. Querying to find the F intersecting pairs takes $O(M \log N + N \log M + F)$ time, inserting and deleting rectangles from active sets takes $O(M \log M + N \log N)$. The overall time is $O((M+N)\log(M+N)+F)$.

LAPIS does not implement the plane-sweep algorithm, so comparing its performance in practice is left for future work.

4.6.8 Counting, Min, and Max

Some of the operator definitions given in Chapter 3 are infeasible to implement directly, since they seem to require large intermediate results (e.g. forming $A - \{a\}$ for every region $a \in A$). This section describes how a few of the more important operators can be implemented efficiently in practice.

The *first* operator returns the first region in a region set in lexicographic order. The definition of this operator given in Section 3.6.5 would be very slow to implement:

$$\textit{first}\, A \equiv \textit{forall}\,(a : A)\,.\, a- > (A - a)$$

In practice, *first* is trivial to implement on a rectangle collection sorted in lexicographic order, like an RB-tree or region array. The result is the lower-left corner of the first rectangle in the collection.

The general counting operator *nth* returns the nth region in a region set. This operator is much harder to optimize, because the lexicographic order of regions does not necessarily correspond to the lexicographic ordering of the rectangles (Figure 4.17). Although the problem could be solved by a plane-sweep technique, LAPIS takes the simpler approach of defining *nth* only for point collections, so that the nth region always corresponds to the nth rectangle. Nested, overlapping, and flat region sets (and intersections, differences, and unions thereof) are always point collections, so this is not a serious limitation. Applying *nth* to a non-point rectangle collection raises an exception in LAPIS.

The *max* and *min* operators return the outermost and innermost regions in a set

$$\begin{aligned} \textit{max}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{in}\,(A - a) \\ \textit{min}\, A &\equiv \textit{forall}\,(a : A)\,.\, a - \textit{contains}\,(A - a) \end{aligned}$$

LAPIS implements *max* by finding the *max* of each rectangle in A, which is just the rectangle's upper-left corner $[s_1, e_2]$, then querying the collection for *contains* $[s_1, e_2]$ to check that no other regions contain it. *Min* is implemented similarly.

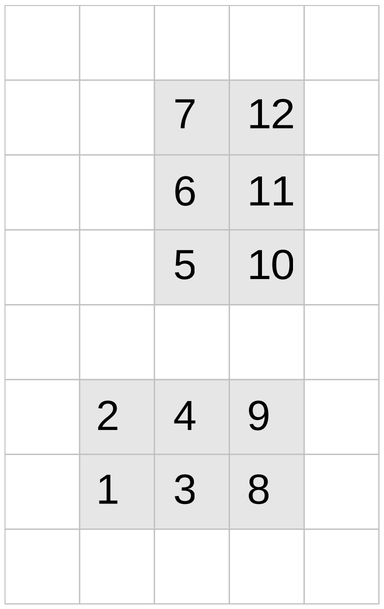


Figure 4.17: The lexicographic ordering of regions has no simple relationship with lexicographic ordering of rectangles. Finding the nth region in a rectangle collection may require jumping back and forth between rectangles.

Data structure	Region sets	Query	Insert	Delete	Space
RB-tree	any	$O(N)$	$O(\log N)$	$O(N+F)$	$O(N)$
quadtree	any	$O(F \log n)$	$O(n)$	$O(F \log n)$	$O(\min(nN, n^2))$
4D-tree	any	$O(N^{3/4}+F)$	$O(\log N)$	$O(\log N + F)$	$O(N)$
4D range tree	any	$O((\log N)^3 + F)$	$O((\log N)^3)$	$O((\log N)^3 + F)$	$O(N(\log N)^3)$
region array	monotonic	$O(\log N + F)$	$O(\log N)$	$O(\log N + F)$	$O(N)$
syntax tree	nested	$O(\log N + F)$	$O(\log N)$	$O(\log N + F)$	$O(N)$

Table 4.1: Rectangle-collection data structures. N is the number of rectangles in the collection, F is the number of rectangles found by a query or affected by a deletion, and n is the length of the string. Boldface indicates that the data structure is implemented in LAPIS.

4.7 Performance

The rectangle-collection data structures and algorithms discussed in this chapter are summarized in Tables 4.1–4.5. A boldfaced entry indicates that the data structure or algorithm is implemented in LAPIS. Some data structures are appropriate only to certain kinds of region sets. The type hierarchy of region sets is summarized in Figure 4.18.

These tables only give the worst-case time for each operation. To study the average case, the performance of the LAPIS implementation was measured directly with two experiments. The first experiment tested each algebra operator on random region sets of increasing size, in order to see how individual operators scale. The second experiment applied real patterns (the patterns from the LAPIS library) to a set of real documents (web pages, source code, and plain text), in order to gain a sense for how region algebra operators perform in practice.

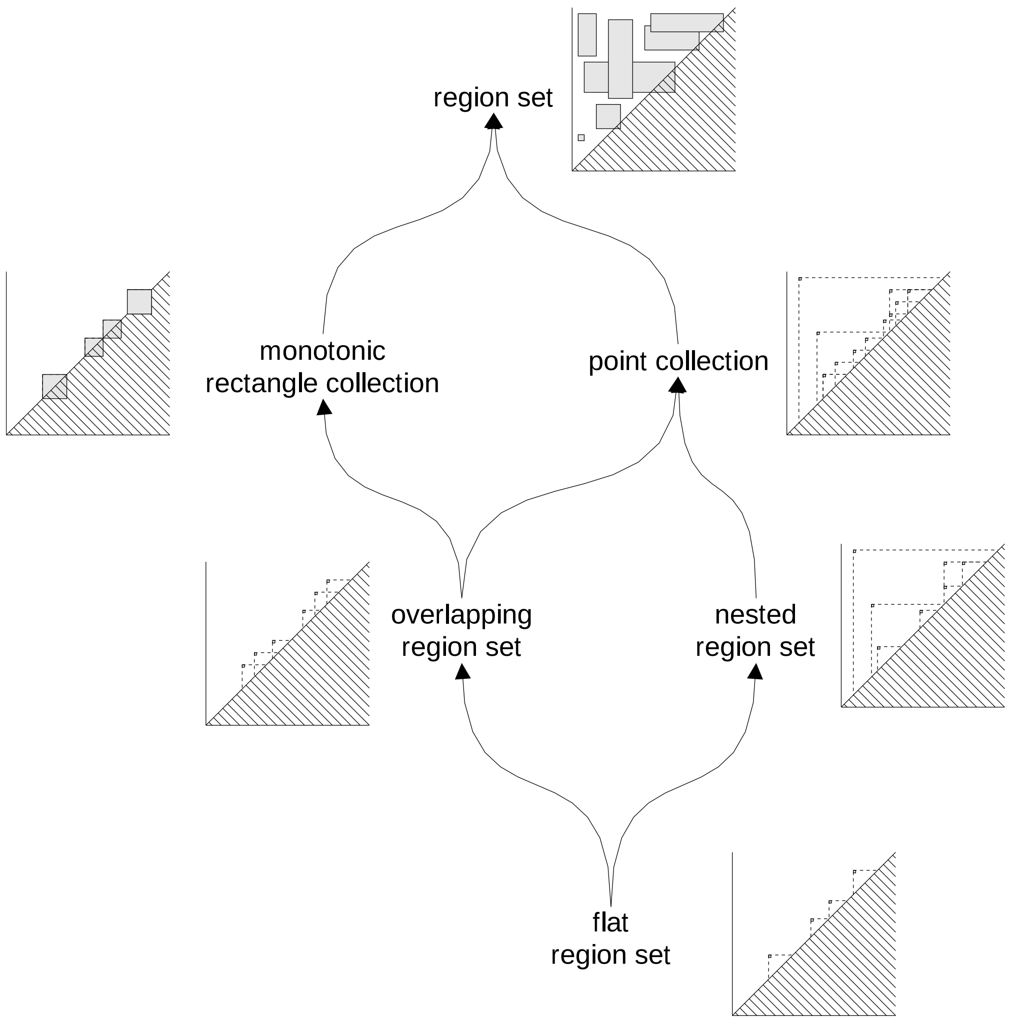


Figure 4.18: Hierarchy of region set types.

Expression	Time	Result
***op* RB-tree**	$O(N \log N)$	RB-tree
op quadtree	$O(\min(nN, n^2))$	quadtree
***op* region array**	$O(N)$	region array

Table 4.2: Unary relational operators, such as *in, contains, just-before, just-after,* etc.

Expression	Time	Result
RB-tree ∩ RB-tree (tandem)	$O(MN + F \log F)$	RB-tree
RB-tree ∩ RB-tree (plane-sweep)	$O((M+N)\log(M+N) + F)$	RB-tree
quadtree ∩ quadtree	$O(\min(nN, n^2))$	quadtree
point RB-tree ∩ region array	$O(M \log N)$	point RB-tree
point RB-tree ∩ point RB-tree	$O(M + N)$	point RB-tree

Table 4.3: Set intersection.

Expression	Time	Result
RB-tree $\cup$ **RB-tree**	$O((M+N)\log(M+N))$	RB-tree
quadtree $\cup$ quadtree	$O(\min(nN, n^2))$	quadtree
point RB-tree $\cup$ point RB-tree	$O(M+N)$	point RB-tree

Table 4.4: Set union.

Expression	Time	Result
RB-tree $-$ **RB-tree**	$O(MN^2)$	RB-tree
quadtree $-$ quadtree	$O(\min(nN, n^2))$	quadtree
point RB-tree $-$ region array	$O(M \log N)$	point RB-tree
point RB-tree $-$ point RB-tree	$O(M+N)$	point RB-tree

Table 4.5: Set difference.

4.7.1 Random Microbenchmarks

The first experiment tested LAPIS against randomly-generated region sets. No documents were needed for this experiment. Since region sets are just sets of integer pairs, they can be generated at random without reference to any document.

Region sets were generated in such a way in such a way that expected number of intersections between any two region sets of size N would be $O(N)$. Two kinds of region sets were generated: flat and nested. Random flat region sets were generated by first choosing $2N$ small lengths l_i uniformly from $0 \ldots 9$, then using the lengths to create a set of $2N$ endpoints $p_i = \sum_{j=1}^{i} l_i$, then connecting each pair of endpoints $[p_{2i}, p_{2i+1}]$ to create a flat set of N regions. Two flat sets of size N generated with this approach would share $N/100$ regions on the average. A random nested region set was generated by first generating $2N$ endpoints as for flat region sets, then proceeding in sorted order through the endpoints, making a uniform decision at each point whether to open a new parent (pushing the point on a stack), close the current parent (popping from the stack), or make a leaf (consuming both the point and its next point to produce a region). When one alternative was impossible (e.g. the stack was empty, or only enough points remained to close the stack), only the possible alternatives were chosen uniformly. This process produced nested region sets that shared $N/200$ regions on the average. Flat region sets were stored in a region array, and nested region sets in an RB-tree.

The timing tests were performed in LAPIS, compiled with Jikes 1.06 and running under Java 1.3.0 with the HotSpot just-in-time compiler, on an 850 MHz Pentium III running Linux. To limit interference from the garbage collector, the tests were run with 128MB of preallocated heap storage, and the garbage collector was forced to run synchronously before every test. Startup effects, such as class loading, just-in-time compiling, and instruction cache warmup, were eliminated by running each sequence of tests twice and keeping only the results of the second. Every test was repeated at least three times or for at least 1 second, whichever took longer, and the average value was reported.

The tests applied each algebra operator to random region sets of varying size. The results are shown in Figures 4.19–4.22. In each graph, the x-axis is the number of rectangles in one operand, N, on a logarithmic scale. The y-axis is the cost per input rectangle, computed by dividing the running time T by the number of rectangles N.

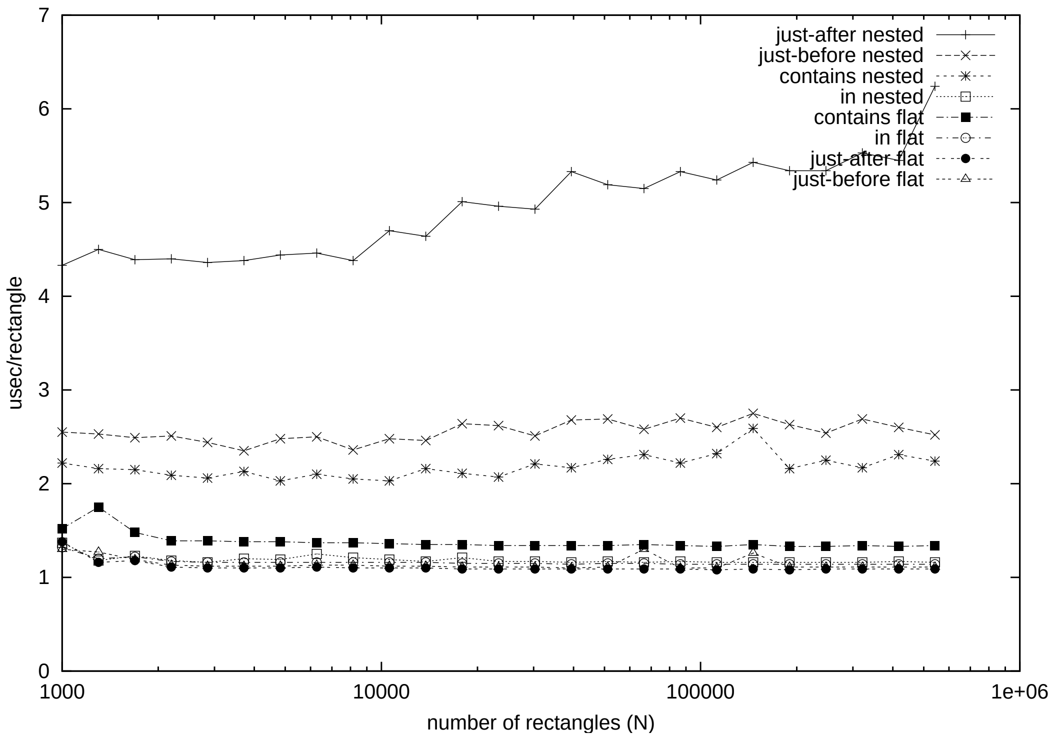


Figure 4.19: Relational operators applied to random region sets of size N.

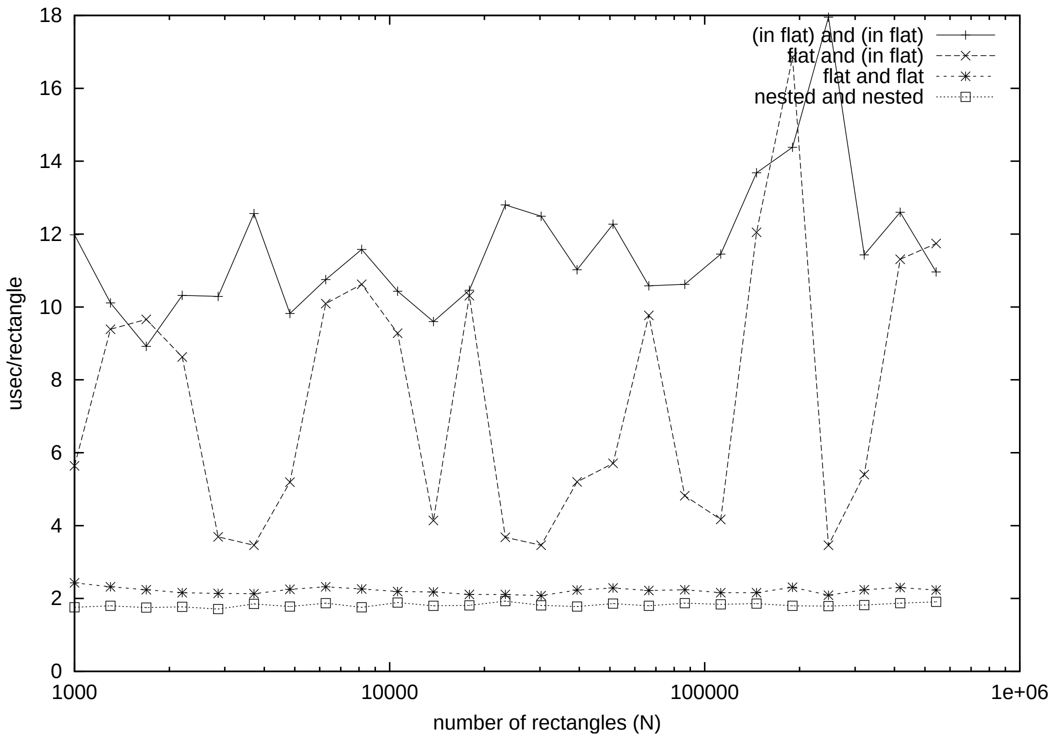


Figure 4.20: Intersection of random region sets of size N.

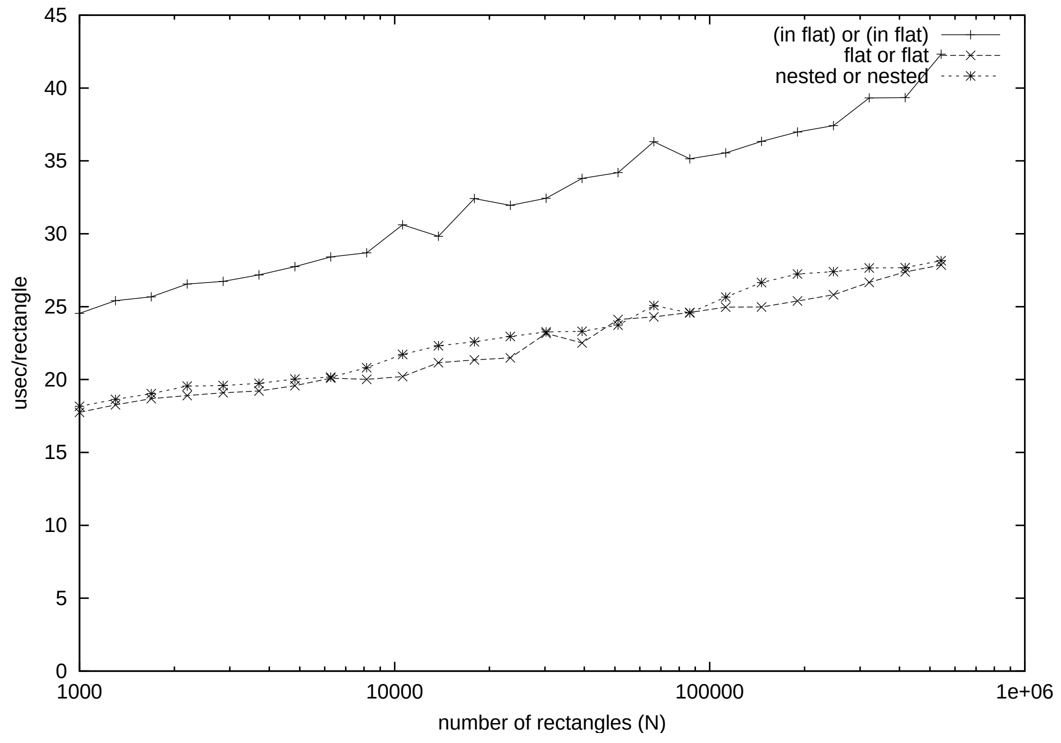


Figure 4.21: Union of random region sets of size N.

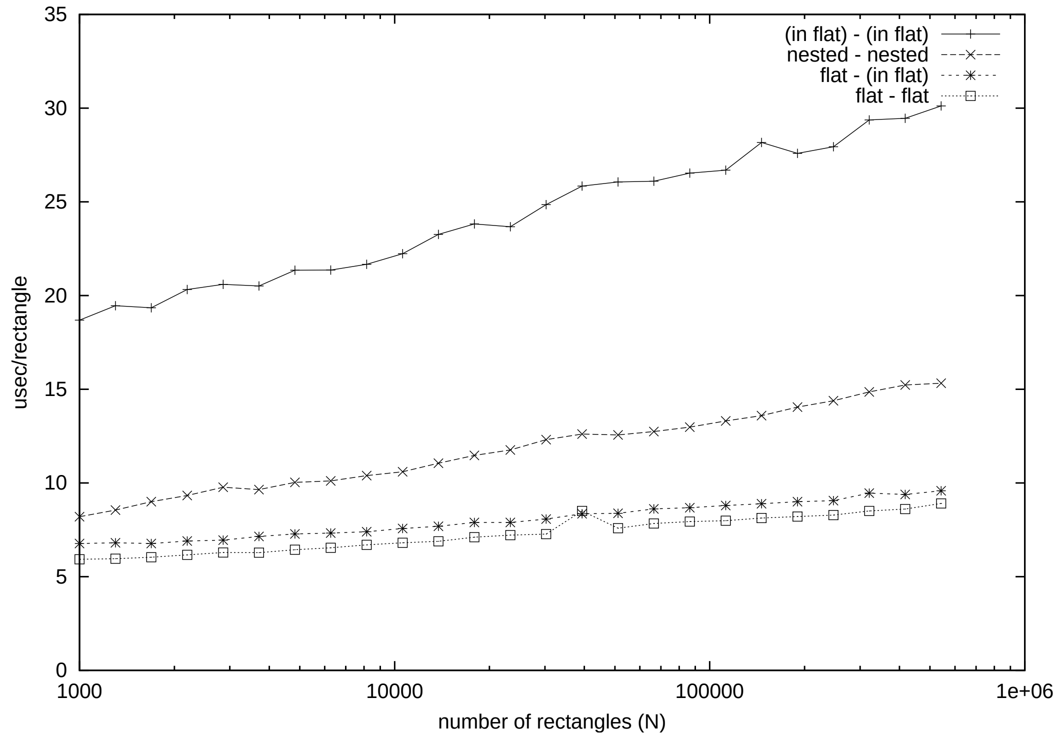


Figure 4.22: Difference of random region sets of size N.

The slope of the line indicates an operator's asymptotic behavior as N increases. When $T = O(N)$, the line is flat ($T/N = k$). When $T = O(N \log N)$, the line has constant slope ($T/N = k \log N$). When $T = O(N^{1+\varepsilon})$ for some $\varepsilon > 0$, the line has an exponential shape ($T/N = k2^{\varepsilon \log N}$).

Figure 4.19 shows some representative relational operators. Most of the relational operators behave like $O(N)$. The only exception is *just-after nested*, which produces its output rectangles in unsorted order, requiring $O(N \log N)$ time to sort them. Relational operators on a flat set (stored in a region array) are considerably faster than relational operators on a nested set (stored in an RB-tree), because the region-array operator produces a simple wrapper around the region array that takes no extra space.

Intersections are more complicated (Figure 4.20). The bottom two lines in the graph (*flat and flat*, *nested and nested*) show that intersections between point collections take linear time. The top two lines (*flat and (in flat)* and *(in flat) and (in flat)*) are much more complicated, because sometimes the result set doesn't require sorting and sometimes it does. The cost per rectangle varies between a constant lower bound and a logarithmic upper bound.

Set union and set difference operators (Figure 4.21 and 4.22) increase like $O(N \log N)$.

4.7.2 Realistic Benchmarks

The second experiment measured the time to evaluate all the TC patterns in the LAPIS library against real documents. The LAPIS library contains 82 TC patterns, which compile into 478 region algebra operators. These TC patterns include concepts like `Number`, `Line`, `Sentence`, `Time`, `Date`, `PhoneNumber`, `ZipCode`, `URL`, and `EmailAddress`, among others. A complete list of the named patterns in the library can be found in Appendix A. The library also contains three parsers (HTML, Java, and Characters), but the running time of the parsers was excluded from the measurements because the parsers are written in Java and do not use the region algebra.

Three sets of documents were used in the test:

- 339 Java source files from the LAPIS source code, ranging in size from 1 KB to 211 KB;
- 158 HTML web pages from the LAPIS distribution and other sources, ranging from 1 KB to 235 KB;
- 63 outputs of `ls -l` for various directories on a Linux workstation, ranging from 1 KB to 134 KB.

For each document, the total time to match all the TC patterns was measured, using the same techniques described in the previous section. The total number of input region rectangles processed by pattern operators was also recorded. The results are shown in Figures 4.23–4.25. In each graph, the x-axis is the total number of rectangles processed by all the patterns, on a logarithmic scale, and the y-axis is the cost per input rectangle, computed by dividing the total running time by the total number of rectangles. Each point represents a single document. The graphs suggest that the typical region algebra operator in these patterns takes between 3 and 6 μsec per rectangle on realistic data, which is in the same ballpark as the microbenchmarks from the previous section. No upward trend is detectable as documents grow larger.

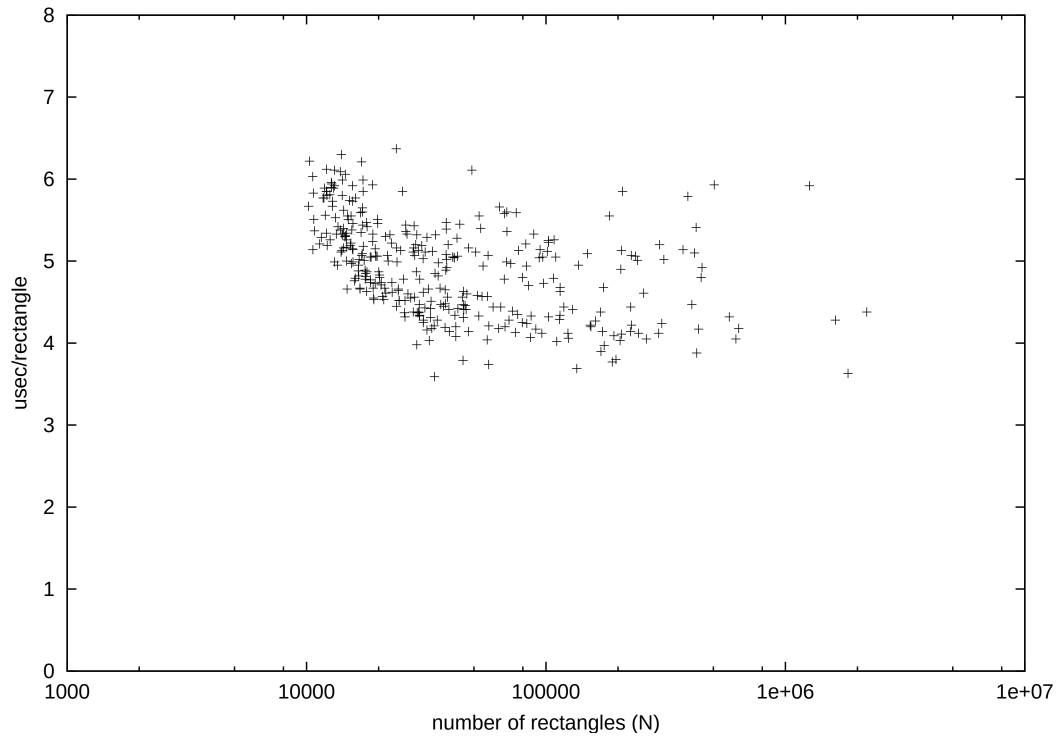


Figure 4.23: Performance of LAPIS library patterns on Java source files.

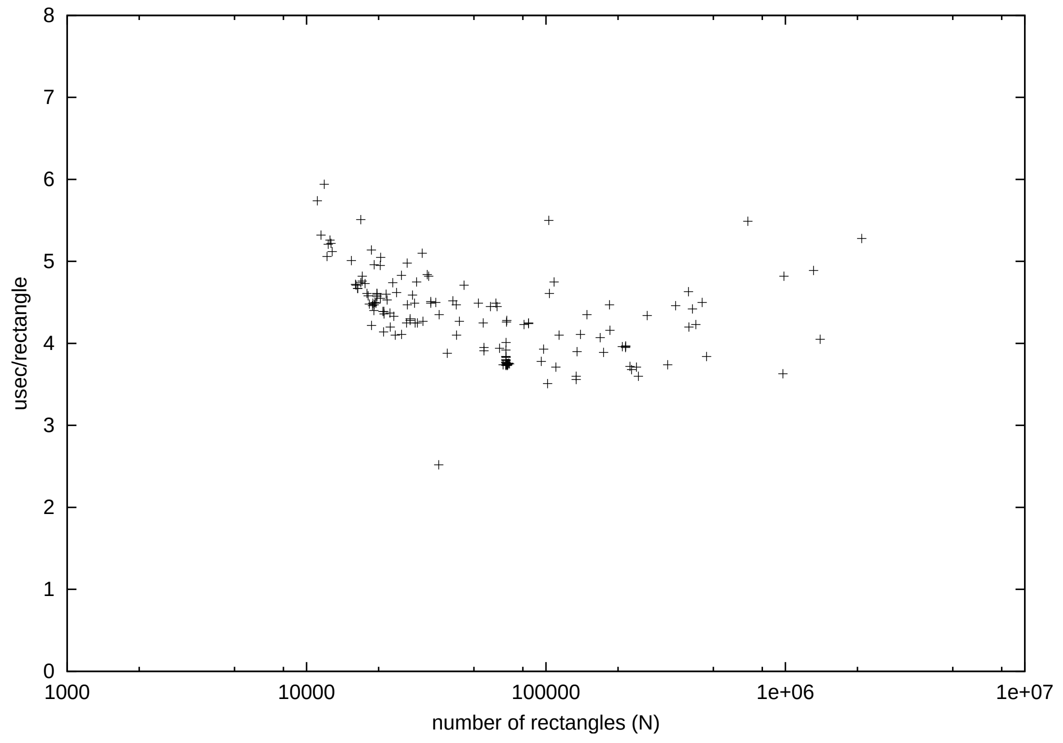


Figure 4.24: Performance of LAPIS library patterns on HTML web pages.

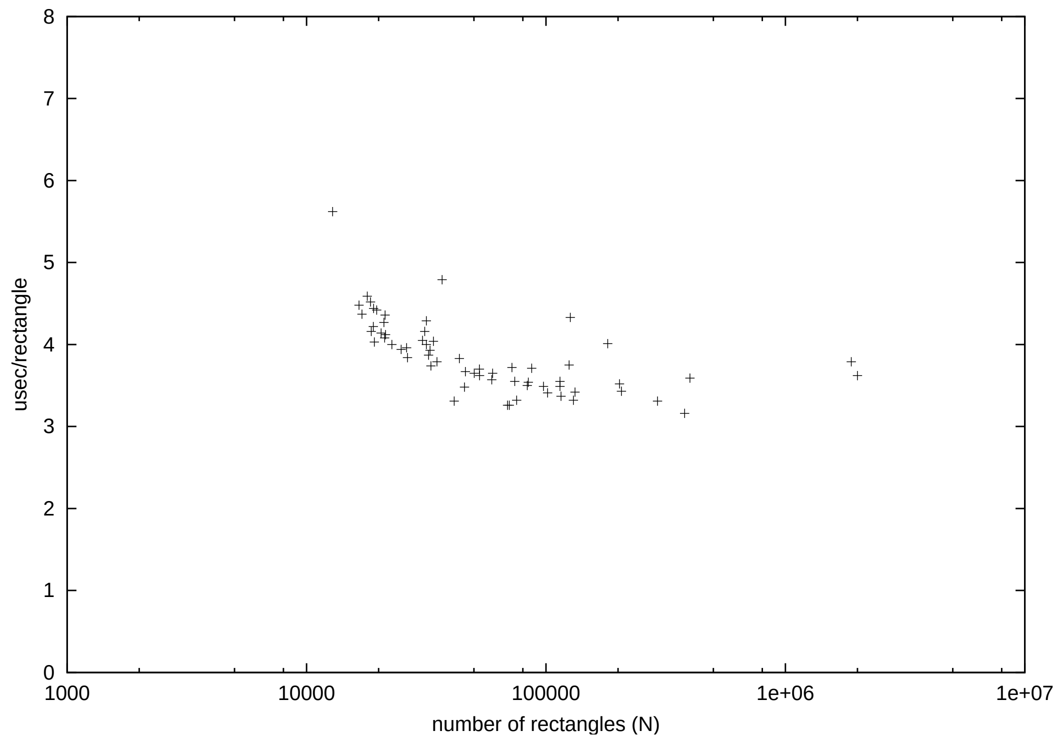


Figure 4.25: Performance of LAPIS library on `ls -l` outputs.

To get a sense for performance in terms of document size, Table 4.6 shows the pattern matching rate for each kind of benchmark document, computed by dividing the size of the document in kilobytes by the total running time on that document in seconds. From the table, a typical library pattern is matched at a rate of roughly 1,500 KB/sec. By comparison, the Jakarta regular expression package [Jak99] matches a simple word pattern (`\w+`) at roughly 750 KB/sec on the same hardware, compiler, and virtual machine. One possible reason for TC's faster performance is that most TC patterns process only a few regions relative to the size of the file, whereas a regular expression tests every character. Another possible reason is that the Jakarta package is not fully optimized, although it is the fastest Java regular expression matcher I have found. It would be interesting to compare the performance of region algebra expressions with regular expressions directly, using equivalent patterns in each language. A proper comparison would require using the fastest regular expression package available, however, which would certainly be written in C or C++. The region algebra implementation would also have to be rewritten in a higher-performance language to make the comparison fair, so this possibility remains future work.

Documents	All patterns in library	Per pattern	Per operator
Java source files	21.5 KB/sec (±2.7)	1,760 KB/sec (±220)	10,300 KB/sec (±1300)
HTML web pages	20.6 KB/sec (±2.0)	1,690 KB/sec (±160)	9,850 KB/sec (±960)
`ls -l` output	17.6 KB/sec (±1.6)	1,440 KB/sec (±130)	8,410 KB/sec (±760)

Table 4.6: Pattern matching rate for the benchmark documents (mean ± standard deviation). *All patterns in library* is the rate at which all TC patterns in the library are matched. *Per pattern* is the average rate for each of the 82 patterns. *Per operator* is the average rate for each of the 478 operators in the patterns.

Chapter 5

Language Theory

One way to study the region algebra is to examine where it fits into the language hierarchy: finite languages, regular languages, context-free languages, etc. This question is complicated by two issues. First, the region algebra describes substrings within a string, not the string itself, so it is necessary to define what is meant by the language recognized by a region algebra expression. Second, the pure region algebra includes only two ground terms, Ω and $\emptyset$. As a result, the class of languages generated by a pure region algebra expression is relatively trivial. In order to make the question relevant and interesting, we have to augment the region algebra with additional ground terms, such as literal strings, regular expressions, and context-free grammars. This is not an artifice, however; it corresponds to the way the region set algebra is designed to be used in practice, to combine text structure detected by other parsers, such as regular expressions and context free grammars.

This chapter establishes the following results:

- The region algebra with only Ω and $\emptyset$ as ground terms (and omitting *forall*) can only recognize the set of all possible regions or the empty set.
- The region algebra with literal strings as ground terms (and omitting *forall*) can recognize a strict subset of regular languages called *noncounting languages* [MP71].
- The region algebra with regular expressions as ground terms can recognize the regular languages.
- The region algebra with context-free grammar nonterminals as ground terms (and omitting *forall*) can recognize a proper superset of the context-free languages, and a subset of the context-sensitive languages. Whether *all* context-sensitive languages can be recognized by the region algebra is still an open question.

5.1 Preliminary Definitions

We begin with some definitions, following the standard conventions used by Hopcroft and Ullman [HU79]. Let Σ be a finite set of symbols, or *alphabet*. Σ^* is the set of all strings over the alphabet Σ, including the empty string ϵ. A *language* L is a subset of Σ^*. A *language class* is a collection of languages.

We now define some familiar language classes. $\mathcal{F}$ is the class of finite languages, i.e. languages L such that the set L is finite. $\mathcal{R}$ is the class of *regular* languages, where a regular language has the following recursive definition:

- $\emptyset$ (the empty language) and $\{\epsilon\}$ (the language containing the empty string) are regular;
- $\{a\}$ for any symbol $a \in \Sigma$ is regular;
- if L and L' are regular, then $L \cup L'$ is regular;
- if L and L' are regular, then $LL' = \{xx' | x \in \text{L and } x' \in L'\}$ is regular;
- if L is regular, then $L^* = \{x_1 \cdots x_n | n \geq 0 \text{ and } \mathrm{x}_i \in L\}$ is regular.
- there are no other regular languages.

$\mathcal{CFL}$ is the class of *context-free* languages, which are defined as follows. A *context-free grammar* is a four-tuple $G = (V, \Sigma, S, P)$ where V is a finite set of nonterminals, Σ is a finite set of terminals ($V \cap \Sigma = \emptyset$) , $S \in V$ is the start symbol, and P is a finite set of productions of the form $A \rightarrow \alpha$, where A is a nonterminal and α is a string of terminals and nonterminals. We write $\alpha \Rightarrow_G \beta$ if the string β can be obtained from the string α by replacing some nonterminal that appears on the left side of a production in G by its right side. We write $\alpha \Rightarrow_G^* \beta$ if β can be obtained from α by zero or more applications of productions. Then the language generated by a context-free grammar is $L(G) = \{x \in \Sigma^* | S \Rightarrow_G^* x\}$. A language is *context-free* if it is generated by some context-free grammar.

Finally, $\mathcal{CSL}$ is the class of *context-sensitive* languages, defined as follows. A *context-sensitive grammar* is a grammar $G = (V, \Sigma, S, P)$ in which productions take the form $\alpha \rightarrow \beta$, where both α and β are strings of terminals and nonterminals and $|\alpha| \leq |\beta|$. We write $\gamma \Rightarrow_G \delta$ if the string δ can be obtained from the string γ by replacing some string α that appears on the left side of a production in G by its right side. Extending $\Rightarrow_G$ to $\Rightarrow_G^*$ in the usual fashion, we say that the language generated by a context-sensitive grammar is $L(G) = \{x \in \Sigma^* | S \Rightarrow_G^* x\}$, and call a language *context-sensitive* if it is generated by some context-sensitive grammar.

It is well-known that $\mathcal{F} \subset \mathcal{R} \subset \mathcal{CFL} \subset \mathcal{CSL}$, where all the inclusions are strict. These relationships are part of the *Chomsky language hierarchy*.

A class of languages is *closed* under some operator if applying the operator to languages in the class always produces another language in the class. Regular languages are closed under union, intersection, and complement. Context-free languages are closed under union, but *not* under intersection or complement. Context-sensitive languages are closed under all three operations [Imm88].

5.2 Finite State Transducers

Several proofs about the power of the region algebra will depend on representing algebra operators as *finite state transducers* (FST), also called generalized state machines. A finite state transducer is a a nondeterministic finite state machine in which every transition is labeled not only by an input symbol, but also by an output string drawn from another, possibly different alphabet. Formally, a finite state transducer is a tuple $M = (Q, \Sigma, \Delta, \delta, q_0, F)$ where Q is a set of states; Σ and Δ are the

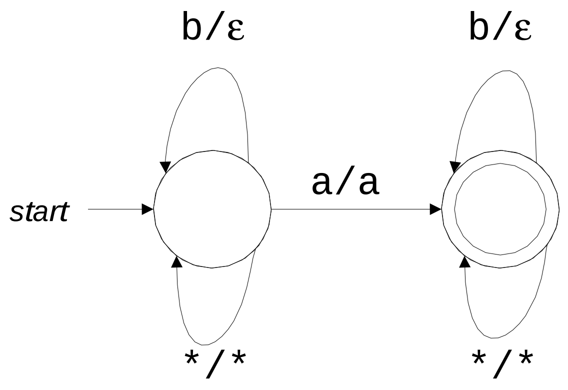


Figure 5.1: A finite state transducer that tests for the letter a and deletes the letter b.

input and output alphabets, respectively; δ is a mapping from $Q \times \Sigma$ to finite subsets of $Q \times \Delta^*$; q_0 is the starting state; and F is the set of accepting states. Note that a transition is labeled with an output *string*, which may be ϵ (the empty string).

Figure 5.1 shows a simple transducer as a state diagram. Transition labels take the form a/w, indicating that the transition occurs (nondeterministically) on input symbol a and emits the output string w, which may be the empty string ϵ. For convenience, state diagrams use $*$ to represent all input symbols that do not already have an explicit transition, and $*/*$ to represent a transition that copies the input symbol to the output. Accepting states are indicated by a double circle. The transducer shown in Figure 5.1 accepts any input string that contains at least one a and emits an output string with all occurrences of b deleted.

A finite state transducer maps an input string x to an output string y if there is some sequence of transitions from q_0 to a final state in F such that the concatenation of the input symbols is x and the concatenation of the output strings is y. Formally, define the extended transition function δ^* mapping $Q \times \Sigma^*$ to $Q \times \Delta^*$ as follows:

$$\begin{array}{rcl} \delta^*(q\,\epsilon) & = & \{(q,\epsilon)\} \\ \delta^*(q,xa) & = & \{(p,w_1w_2\}|\exists p'.(p',w_1) \in \delta^*(q,x) \text{ and } (p,w_2) \in \delta(p',a)\} \end{array}$$

If L is a language over Σ, the mapping $M(L)$ is the language over Δ defined by

$$M(L) = \{y|(p,y) \in \delta^*(q_0,x) \text{ for some } x \in L \text{ and } p \in F\}$$

For our purposes, an important property of finite state transducers is that R and $\mathcal{CFL}$ are closed under mapping through an FST:

Theorem 3. *If L is a regular (or context-free) language and M is any finite state transducer, then $M(L)$ is also regular (or context-free).*

A proof of Theorem 3 is given by Hopcroft and Ullman [HU79, Theorem 11.1].

Context-sensitive languages are closed under a more constrained form of finite state transducers. An FST is *ϵ-free* if no transition has an output string ϵ. Then Hopcroft and Ullman also prove [HU79, Theorem 11.1]:

Theorem 4. *If L is context-sensitive and M is an ϵ-free finite state transducer, then $M(L)$ is context-sensitive.*

5.3 Languages Recognized by a Region Expression

Recall that a region is identified by its start and end offsets, so the region y in the string xyz is described by the pair $[|x|, |x| + |y|]$. For any region expression E and any string w, let E/w be the set of regions produced by applying the expression E to w. For example, "i" *just-before* "s" $/mississippi = \{[1, 2], [4, 5]\}$. Note that the set of regions is a set of locations in w, not a set of strings, so E/w is not itself a language. The question now is how to translate from regions to strings, so that the region algebra can be related to the Chomsky language hierarchy.

Several languages might reasonably be associated with a region expression E:

- The *inner language* $L_0(E)$ is the set of substrings that correspond to the regions matching E:

  $$L_0 = \{y|\exists y, z.[|x|, |x| + |y|] \in E/xyz\}$$

  The inner language captures the matches themselves, but completely omits the context. For example, the inner language of "i" *just-before* "s" consists of just one string, i, which is the same as the inner language of the (semantically quite different) expression "i".

- The *outer language* $L_1(E)$ is the set of strings w that have at least one match to the expression E:

  $$L_1(E) \equiv \{w|E/w \neq \emptyset\}$$

  The outer language represents the context of the matches, but not the locations of the matches themselves. For example, the outer language of "i" *just-before* "s" is the regular language $\Sigma^*\{is\}\Sigma^*$, which is the same as the outer language of the expression "s" *after* "i".

Since neither language fully captures the semantics of the region algebra, it is also useful to define a language that explicitly indicates the location of a region in context. This language consists of strings over the extended alphabet $\Sigma \cup \{[,]\}$, where the square brackets represent unique delimiters that do not occur in the underlying alphabet Σ. The square brackets are used to delimit the substring that matches the region expression.

The *region language* for a region expression $L(E)$ is therefore defined as

$$L(E) \equiv \{x[y]z|[|x|, |x| + |y|] \in E/xyz\}$$

For example, $L($"i" *just-before* "s"$) = \Sigma^*\{[i]s\}\Sigma^*$. Strings in this language include $m[i]ssissippi$, $miss[i]ssippi$, $[i]s$, and $xyz[i]spdq$.

The region language is stronger than the inner and outer languages, in the sense that both the inner language and the outer language can be obtained from the region language. The inner language is the set of strings inside the square brackets: $L_0(E) = \{y|x[y]z \in L(E)\}$. The outer language is the region language with the square brackets removed: $L_1(E) = \{xyz|x[y]z \in L(E)\}$.

The region language is also stronger in the sense that its membership in a language class implies that the inner and outer languages belong to the same class:

Theorem 5. *If $L(E)$ is regular (or context-free), then $L_0(E)$ and $L_1(E)$ are also regular (or context-free).*

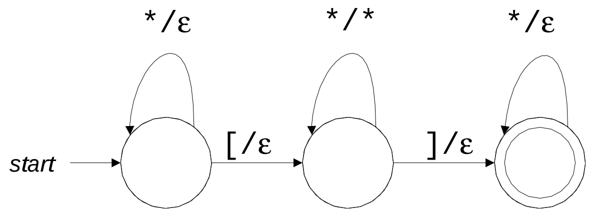


Figure 5.2: Finite state transducer M_0 maps a region language $L(E)$ to the corresponding inner language $L_0(E)$ by erasing symbols outside the square brackets and the square brackets themselves.

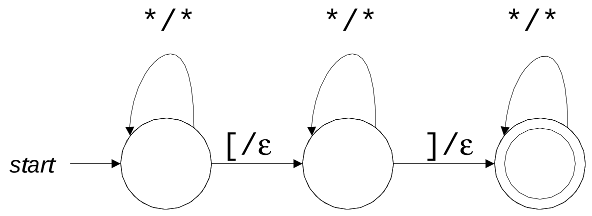


Figure 5.3: Finite state transducer M_1 maps a region language $L(E)$ to the corresponding outer language $L_1(E)$ by erasing just the square brackets.

Proof. We will exhibit a finite state transducer M_0 that maps $L(E)$ into $L_0(E)$, and another transducer M_1 that maps $L(E)$ into $L_1(E)$. Thus, by Theorem 3, if $L(E)$ is regular (or context-free), then $L_0(E)$ and $L_1(E)$ are also regular (or context-free). The M_0 transducer is shown in Figure 5.2. It accepts any input string with a square-bracketed region, such as $x[y]z$, and emits only the part in brackets, y. This has the effect of mapping a region language $L(E)$ to the corresponding inner language $L_0(E)$. The M_1 transducer, shown in Figure 5.3, simply erases the square brackets, transforming a string like $x[y]z$ into xyz. It should be clear that this maps the region language $L(E)$ to its outer language $L_1(E)$. $\square$

The converse of Theorem 5 is not necessarily true, however, because the inner and outer languages do not contain enough information. Consider the following grammar:

$$S \quad \rightarrow \quad aSb \,|\, \epsilon$$

This grammar matches a context-free language of the form $a^n b^n$. Let E be the expression $a^* b^*$ *in* S. Then the inner language $L_0(E)$ is simply the regular language $a^* b^*$, but the region language $L(E)$ is $a^i[a^{n-i}b^j]b^{n-j}$, which is not regular but context-free.

A similar example shows regularity of the outer language does not imply regularity of the region language. Consider the grammar

$$\begin{aligned} S \quad &\rightarrow \quad aS \,|\, Sb \,|\, B \\ B \quad &\rightarrow \quad aBb \,|\, \epsilon \end{aligned}$$

Let E be the expression B *in* S. The outer language $L_1(E)$ is the set of all strings recognized by the entire grammar, which is actually the regular language a^*b^*. The region language $L(E)$, however, corresponds to the language $a^*[a^nb^n]b^*$, which is not regular but context-free.

5.4 Restricted Algebras

In order to simplify the presentation, it will be helpful to restrict discussion to certain subsets of the algebra operators. such as relational or set operators. Each subset is denoted by a letter:

- R: the relational operators *before, after, in, contains, overlaps-start,* and *overlaps-end*.
- S: the set operators $\cup, \cap$, $-$, Ω, and $\emptyset$.
- A: the *forall* operator.

The most difficult operator to deal with semantically is *forall*, because it binds variables. However, the region algebra presentation in Chapter 3 depended on *forall* to define *then* (string concatenation), which is a fundamental operator in other language formalisms like regular expressions and context-free grammars. When we want to avoid the difficulties of *forall* but still have access to a concatenation operator, we will treat *then* as if it were a primitive operator:

- T: the operator *then.*

Region algebra expressions may also include *ground terms* (nullary operators). We consider four kinds of ground terms in this chapter:

- $\emptyset$: no ground terms (other than the set operators Ω and $\emptyset$).
- $\mathcal{F}$: strings in a finite language (i.e., literal strings) may be used as ground terms.
- $\mathcal{R}$: regular expressions may be used as ground terms.
- $\mathcal{CFL}$: nonterminals of a context-free grammar may be used as ground terms.

Note that each class of ground terms is a superset of the previous class: regular languages include finite languages as a special case, and context-free languages include regular languages.

We use these letters to name a subset of the region algebra. The permitted operators are listed, followed by a subscript indicating the permitted ground terms. For example:

- $RS_\emptyset$ is the class of region algebra expressions using only relational operators (R), set operators (S), and no ground terms other than Ω and $\emptyset$.
- $RST_\mathcal{F}$ is the class of expressions using relational operators, set operators, concatenation (*then*), and literal strings.
- $RSTA_\mathcal{R}$ is the class of expressions using any algebra operator and regular expressions as ground terms. Since *then* can be defined in terms of *forall*, this class can also be written $RSA_\mathcal{R}$. I prefer to write $RSTA$, since it makes it clearer that $RSTA$ is a superset of RST.

Obviously, $RS_G \subset RST_G \subset RSTA_G$ for any choice of ground terms G, and $P_\emptyset \subset P_\mathcal{F} \subset P_\mathcal{R} \subset P_{\mathcal{CFL}}$ for any choice of operators P.

Not all restrictions will be interesting. This chapter will focus on RST (the region algebra without *forall*) and $RSTA$ (with *forall*), for all the choices of ground terms.

5.5 Algebra Semantics

We can now define the region algebra operators in terms of the region languages they recognize.

Set Operators (S)

The set operators are straightforward. For example, $x[y]z$ is in the language $L(E_1 \cap E_2)$ if and only if it is in both $L(E_1)$ and $L(E_2)$.

$$\begin{aligned} L(E_1 \cap E_2) &= L(E_1) \cap L(E_2) \\ L(E_1 \cup E_2) &= L(E_1) \cup L(E_2) \\ L(E_1 - E_2) &= L(E_1) - L(E_2) \end{aligned}$$

The nullary set operators are defined as follows:

$$\begin{aligned} L(\Omega) &= \{x[y]z | x, y, z \in \Sigma^*\} \\ L(\emptyset) &= \emptyset \end{aligned}$$

Relational Operators (R)

The relational operators are similarly straightforward. For example, if $x[y]z$ is in $L(E)$, then for all u, v, w such that $x = uvw$, we have $u[v]wyz$ in $L(\textit{ before } E)$.

$$\begin{aligned} L(\textit{ before } E) &= \{v[w]xyz | vwx[y]z \in L(E)\} \\ L(\textit{ after } E) &= \{vwx[y]z | v[w]xyz \in L(E)\} \\ L(\textit{ in } E) &= \{vw[x]yz | v[wxy]z \in L(E)\} \\ L(\textit{ contains } E) &= \{v[wxy]z | vw[x]yz \in L(E)\} \\ L(\textit{ overlaps-start } E) &= \{v[wx]yz | vw[xy]z \in L(E)\} \\ L(\textit{ overlaps-end } E) &= \{vw[xy]z | v[wx]yz \in L(E)\} \end{aligned}$$

Then Operator (T)

Since we will need *then* as a primitive operator, we define it:

$$L(E_1 \textit{ then } E_2) = \{w[xy]z | w[x]yz \in L(E_1) \wedge wx[y]z \in L(E_2)\}$$

Following the convention in the language theory literature, the expressions in this chapter will omit the explicit *then*, representing the concatenation of two region expressions by simple juxtaposition:

$$E_1 E_2 \equiv E_1 \textit{ then } E_2$$

Literal Strings ($\mathcal{F}$)

The region language corresponding to a literal string expression w is the set of all strings that contain w in square brackets:

$$L(x) = \Sigma^* \{[w]\} \Sigma^*$$

Regular Expressions ($\mathcal{R}$)

The region language corresponding to a regular expression R is the set of all strings that contain a string recognized by R in square brackets:

$$L(R) = \Sigma^* \{[\} L(R) \{]\} \Sigma^*$$

Context-Free Grammar Nonterminals ($\mathcal{CFL}$)

Suppose a context-free grammar G has start symbol S and some nonterminal A. Then

$$L(A) = \{x[y]z | S \Rightarrow_G^* xAz \wedge xAz \Rightarrow_G^* xyz\}$$

These definitions are sufficient to prove results about the region algebra without the *forall* operator: $RST_\emptyset$, $RST_\mathcal{F}$, $RST_\mathcal{R}$, and $RST_\mathcal{CFL}$. Discussion of *forall* is postponed to a later section, since it requires some extra machinery to define its semantics.

5.6 Trivial Ground Terms ($RST_\emptyset$)

We first dispense with a trivial class, $RST_\emptyset$. Algebra expressions of this class can only use Ω and $\emptyset$ as ground terms. Since such an expression has no way to test the symbols of the string, it must either accept *all* possible region languages ($L(\Omega) = \{x[y]z | x, y, z \in \Sigma^*\}$) or none at all ($L(\emptyset) = \emptyset$). Thus the region language class recognized by $RST_\emptyset$ contains precisely two languages, $L(\Omega)$ and $L(\emptyset)$.

Theorem 6. *The region languages recognized by $RST_\emptyset$ are $L(\Omega)$ and $L(\emptyset)$.*

Proof. Proof by structural induction on an algebra expression E.

- Ground terms: only Ω and $\emptyset$ are allowed, so an expression consisting of just one term recognizes either $L(\Omega)$ or $L(\emptyset)$.

- Relational operators: If $L(E) = \emptyset$, then $L(\textit{before}\, E) = \emptyset$, trivially. So suppose $L(E) = L(\Omega)$. For any string xyz, we must have $xyz[] \in L(E)$, so $x[y]z \in L(\textit{before}\, E)$. Since xyz was arbitrary, $L(\textit{before}\, E) = L(\Omega)$. Similar arguments work for *after, in, contains, overlaps-start*, and *overlaps-end*.

- Set operators: the intersection, union, or difference of any combination of $\{L(\Omega), L(\emptyset)\}$ can only be $L(\Omega)$ or $L(\emptyset)$.

- Concatenation: if either $L(E_1)$ or $L(E_2)$ is empty, then $L(E_1E_2)$ is empty. So suppose $L(E_1) = L(E_2) = L(\Omega)$. For any string xyz, we must have $x[y]z \in L(E_1)$ and $xy[]z \in L(E_2)$, so $x[y]z \in L(E_1E_2)$. Since xyz was arbitrary, $L(E_1E_2) = L(\Omega)$.

$\square$

Corollary 2. *The only inner languages and outer languages recognized by $RST_{\emptyset}$ are Σ^* and $\emptyset$.*

This theorem shows that the region algebra is of little use by itself. It depends on other ground terms, such as literal strings, regular expressions, or context-free grammars, to recognize useful languages.

5.7 Finite Ground Terms ($RST_{\mathcal{F}}$)

The next interesting specialization of the region algebra is $RST_{\mathcal{F}}$, which adds literal strings (including the empty string ϵ) to $RST_{\emptyset}$. This algebra is more powerful than it looks at first blush. Suppose the alphabet Σ is $\{a, b\}$. The inner language of a simple expression like a *in* aba is the finite language $\{a\}$, but the inner language of the expression $\neg$ *contains* "b" is the infinite language a^*. (Recall that the unary complement $\neg E$ is equivalent to the set difference $\Omega - E$.) Another interesting example is

$$(a\Omega) \cap (\Omega b) \cap (\neg\, \textit{contains}\, aa) \cap (\neg\, \textit{contains}\, bb)$$

whose inner language is $a(ba)^*b$.

The use of *contains* in these examples is not strictly necessary, because *contains* E is equivalent to $\Omega E \Omega$. For simplicity, then, we will begin by ignoring the relational operators and considering the class $ST_{\mathcal{F}}$ (set operators, concatenation, and literal strings). $ST_{\mathcal{F}}$ is the region algebra equivalent of a class of languages that have been well studied, namely *star-free regular expressions* [MP71]. As the name suggests, a star-free regular expression does not use the Kleene star operator *, but it may use intersection and complement as well as concatenation and union. Formally:

Definition 1. *A* star-free regular expression *has the following recursive definition:*

- *$\emptyset$ (the empty set), ϵ (the empty string), and a for any $a \in \Sigma$ are star-free regular expressions;*

- *if R and R' are star-free regular expressions, then $R \cup R'$, $R \cap R'$, $\neg R$, and RR' are star-free regular expressions;*

- *there are no other star-free regular expressions.*

For convenience, and to parallel the region algebra, we will use Ω in star-free regular expressions to represent the expression $\neg\emptyset$. Note that Ω matches the set of all possible strings Σ^* without having to resort to a star operator.

5.7.1 Noncounting Languages (NC)

McNaughton and Papert [MP71] show that the star-free regular expressions recognize a class of languages they call NC, or "noncounting languages". NC is a strict subset of the regular languages. The intuition behind noncounting languages is that a noncounting language must be recognized without needing to count modulo an integer ≥ 2. For example, $(aaa)^*$ is not in NC, because it requires testing whether the length of a string of a's is a multiple of 3. The language $a^*ba^*ba^*$ is in *NC*, however. Although it requires counting the number of b's and testing that the count is exactly 2, this test does not require counting modulo an integer. We can readily find a star-free regular expression for this language: $(\neg(\Omega b\Omega))b(\neg(\Omega b\Omega))b(\neg(\Omega b\Omega))$.

Formally, the language class NC is defined as follows:

Definition 2. *A language L is* noncounting *if and only if for some $n > 0$ and all strings $u, v, w \in \Sigma^*$, $uv^nw \in L$ if and only if $uv^{n+x}w \in L$ for all positive integers x.* NC *is the class of noncounting languages.*

Using this definition, we can show why $(aaa)^*$ is not in NC. Suppose there is some n that satisfies the definition for $(aaa)^*$, and take $u = \epsilon$, $v = a$, and $w = \epsilon$. If $uv^nw = a^n$ is in the language, then $uv^{n+1}w = a^{n+1}$ is *not* in the language. Conversely, if uv^nw is not in the language, then either $uv^{n+1}w$ or $uv^{n+2}w$ *is* in the language. By contradiction, then, no n exists that can make $(aaa)^*$ a noncounting language. Thus NC is a strict subset of the regular languages R.

To illustrate how the definition of NC will be used in proofs, let us prove the analog of Theorem 5 for noncounting languages.

Theorem 7. *For any algebra expression E, if the region language $L(E)$ is noncounting, then the inner language $L_0(E)$ and the outer language $L_1(E)$ are also noncounting.*

Proof. Since $L(E)$ is noncounting, there exists an integer n such that $uv^nw \in L$ if and only if $uv^{n+x}w \in L$ for all strings u, v, w and positive integers x. The proof will first show that the inner language $L_0(E)$ satisfies the same property. If $uv^nw \in L_0(E)$, then $p[uv^nw]q \in L(E)$ for some $p, q \in \Sigma^*$. We can pump this string to get $p[uv^{n+x}w]q \in L(E)$ for any integer x, which implies that $uv^{n+x}w \in L_0(E)$. A similar argument shows that $uv^nw \notin L_0(E)$ implies $uv^{n+x}w \notin L_0(E)$. Since $u, v, w,$ and x were arbitrary, we have shown that $L_0(E)$ is noncounting.

Proving that the outer language $L_1(E)$ is noncounting is slightly harder, but the techniques used will come in handy for the proofs in the next section. We will show that $L_1(E)$ satisfies Definition 2 for the integer $3n$, rather than n. Suppose we are given a string $uv^{3n}w \in L_1(E)$. Since this string is in the outer language of E, there must be some string in the region language of E that differs only in that it has a left and right bracket inserted around the matching region; i.e., there must be some $p[q]r \in L(E)$ such that $pqr = uv^{3n}w$. Any way of splitting $uv^{3n}w$ into three strings p, q, and r must leave at least v^n as a substring of either p, q, or r. Assume without loss of generality that p contains v^n, so that $p = p_1v^np_2$. Then we can pump $p_1v^np_2[q]r$ to get $p_1v^{n+x}p_2[q]r \in L(E)$, so $p_1v^{n+x}p_2qr = uv^{3n+x}w \in L_1(E)$. A similar argument shows that $uv^nw \notin L_1(E)$ implies $uv^{n+x}w \notin L_1(E)$, so $L_1(E)$ is noncounting. □

5.7.2 $RST_{\mathcal{F}} = NC$

We are now ready to prove that $RST_{\mathcal{F}}$ has the same power as star-free regular expressions. First we show that the region languages recognized by $RST_{\mathcal{F}}$ are noncounting languages.

Theorem 8. *If L is the region language of an expression in $RST_{\mathcal{F}}$, then L is noncounting.*

Proof. By structural induction on the algebra expressions in $RST_{\mathcal{F}}$.

- Ground terms. The region languages of all ground terms can be described by star-free regular expressions: $L(\Omega)$ is $\Omega[\Omega]\Omega$, $L(\emptyset)$ is $\emptyset$, and $L(w)$ is $\Omega[w]\Omega$.

- Set operators. NC is closed under intersection, union, and complement (since star-free expressions are likewise closed), so if $L(E_1)$ and $L(E_2)$ are noncounting, then so are $L(E_1 \cap E_2)$, $L(E_1 \cup E_2)$, and $L(E_1 - E_2)$.

- Concatenation. Suppose $L(E_1)$ and $L(E_2)$ are noncounting. Then there exist integers $n_1 > 0$ and $n_2 > 0$ satisfying Definition 2 for $L(E_1)$ and $L(E_2)$, respectively. Let $n = \max(n_1, n_2)$. The claim is that $2n$ satisfies Definition 2 for the language $L(E_1E_2)$. We need to show that for any strings u, v, w, $uv^{2n}w \in L(E_1E_2)$ if and only if $uv^{2n+x}w \in L(E_1E_2)$ for any $x > 0$. Since $uv^{2n}w$ is a string in a region language, it must contain exactly one left and one right square bracket, in that order. Neither bracket can fall in v^{2n}, which leaves only three cases:

  1. Both brackets in u. Write $uv^{2n}w = u_1[u_2]u_3v^{2n}w$ for some u_1, u_2, u_3. Then $u_1[u_2]u_3v^{2n}w \in L(E_1E_2)$ if and only if $u_1[r]su_3v^{2n}w \in L(E_1)$ and $u_1r[s]u_3v^{2n}w \in L(E_2)$ for some $rs = u_2$. Since $L(E_1)$ and $L(E_2)$ are in NC, we have $u_1[r]su_3v^{2n+x}w \in L(E_1)$ and $u_1r[s]u_3v^{2n+x}w \in L(E_2)$, which is true if and only if $uv^{2n+x}w \in L(E_1E_2)$.
  2. Both brackets in w. Symmetric to case 1.
  3. Left bracket in u, right bracket in w. Write $uv^{2n}w = u_1[u_2v^{2n}w_1]w_2$. Then $u_1[u_2v^{2n}w_1]w_2 \in L(E_1E_2)$ if and only if for $u_1[r]sw_2 \in L(E_1)$ and $u_1r[s]w_2 \in L(E_2)$ for some $rs = u_2v^{2n}w_1$. Any way of splitting the region must leave at least v^n as a substring of either r or s. Assume without loss of generality that r contains v^n, so $r = u_2v^nt$ for some t. Then we can pump v^n to v^{n+x} because $L(E_1)$ is in NC: $u_1[u_2v^{n+x}t]sw_2 \in L(E_1)$, so $uv^{2n+x}w \in L(E_1E_2)$.

- Relational operators. The proofs for relational operators are similar to the proof for concatenation. In each case, $L(E)$ is assumed to be noncounting by the induction hypothesis, and n is the integer satisfying satisfying Definition 2 for $L(E)$. We show that $3n$ satisfies Definition 2 for $L(\textit{op}\, E)$.

  – *before*: For $uv^{3n}w \in L(\textit{before}\, E)$, the crucial case is $uv^{3n}w = u_1[u_2]u_3v^{3n}w$. The other two cases are trivial, like cases 1 and 2 for concatenation. $u_1[u_2]u_3v^{3n}w \in L(\textit{before}\, E)$ if and only if $u_1u_2r[s]t \in L(E)$ for some $rst = u_3v^{3n}w$. Any way of splitting $u_3v^{3n}w$ into three strings r, s, t must leave at least v^n as a substring of r, s, or t, so we can pump v^n to v^{n+x}, giving $uv^{3n+x}w \in L(\textit{before}\, E)$.

  – *after*: symmetric with *before*.

  – *overlaps-start*: There are two nontrivial cases. When $uv^{3n}w = u_1[u_2]u_3v^{3n}w$, the string is in $L(\textit{overlaps-start}\, E)$ if and only if $u_1r[s]t \in L(E)$ for some $rst = u_2u_3v^{3n}w$ (where $|r| < |u_2|$ and $|rs| > |u_2|$). Any way of splitting $u_2u_3v^{3n}w$ into three strings

r, s, t must leave at least v^n as a substring of s or t, so we can pump v^n to v^{n+x}, giving $uv^{3n+x}w \in L(\textit{overlaps-start}\, E)$. For the other hard case, where $uv^{3n}w = u_1[u_2v^{3n}w_1]w_2$, we must have $u_1r[s]t \in L(E)$ for some $rst = u_2u_3v^{3n}w$. Again, any way of forming r, s, t must leave at least v^n as a substring of s or t, so we can pump v^n to v^{n+x} and get $uv^{3n+x}w \in L(\textit{overlaps-start}\, E)$

– *overlaps-end*: symmetric with *overlaps-start*.
– *in*: There are two nontrivial cases. When $uv^{3n}w = u_1[u_2]u_3v^{3n}w$, the string is in $L(\textit{in}\, E)$ if and only if $p[qu_2r]s \in L(E)$ for some $pq = u_1$ and $rs = u_3v^{3n}w$. Any way of forming r and s must leave at least v^n in one of them, so we can pump to get $uv^{3n+x}w \in L(\textit{in}\, E)$. The other hard case, when $uv^{3n}w = uv^{3n}w_1[w_2]w_3$, is symmetric.
– *contains*: Only one nontrivial case: when $uv^{3n}w = u_1[u_2v^{3n}w_1]w_2$, the string is in $L(\textit{contains}\, E)$ if and only if $u_1r[s]tw_2 \in L(E)$ for some $rst = u_2v^{3n}w_1$. Any way of forming r, s, and t must leave at least v^n in one of them, so we can pump to get $uv^{3n+x}w \in L(\textit{contains}\, E)$.

□

Corollary 3. *The inner and outer languages of an expression in $RST_{\mathcal{F}}$ are noncounting.*

The previous theorem showed that every language recognized by $RST_{\mathcal{F}}$ is noncounting. It is trivial to show the converse, that every noncounting language is recognized by some expression in $RST_{\mathcal{F}}$.

Theorem 9. *If L is noncounting, then L is the inner language of some expression in $RST_{\mathcal{F}}$.*

Proof. There is some star-free regular expression that recognizes L. Every star-free regular expression corresponds directly to an algebra expression in $ST_{\mathcal{F}}$ whose inner language is the same, and every algebra expression in $ST_{\mathcal{F}}$ is also in $RST_{\mathcal{F}}$. □

5.8 Regular Ground Terms ($RST_{\mathcal{R}}$)

We now move up to a stronger set of ground terms: regular expressions. Note that allowing regular expressions as *ground terms* in the region algebra does not mean that we allow regular expression operators anywhere in the expression. In particular, the Kleene star operator * is not permitted to have any non-regular operators, such as $\cap$, $-$, or *in*, in its scope. The other regular expression operators, union and concatenation, are also algebra operators in $RST_{\mathcal{R}}$, so this restriction does not apply to them. For example, these expressions are in $RST_{\mathcal{R}}$:

$$a^* \textit{ before } b^*$$
$$(a \cup b)^* \textit{ in } (a^*b^*)^*$$

but these are not:

$$(a \textit{ before } ab)^*$$
$$(a\Omega \cap \Omega b)^*$$

This restriction mirrors the LAPIS implementation, which uses regular expressions only as ground terms. It would be interesting to make the Kleene star a full-fledged region algebra operator — and, in fact, the theorems proved in this section imply that the Kleene star would not change the recognition power of $RST_{\mathcal{R}}$, since it is regular — but its precise definition and implementation is left as future work.

$RST_{\mathcal{R}}$ trivially recognizes all regular languages, because it includes regular expressions as a subset:

Theorem 10. *If L is regular, then L is the inner language of some expression in $RST_{\mathcal{R}}$.*

The harder proof goes in the other direction: that every language recognized by $RST_{\mathcal{R}}$ is regular.

Theorem 11. *If L is the region language for some expression in $RST_{\mathcal{R}}$, then L is regular.*

Proof. As usual, the proof proceeds by structural induction on algebra expressions in $RST_{\mathcal{R}}$.

- Ground terms. The ground terms all produce regular sets: $L(\Omega)$ is $\Sigma^*[\Sigma^*]\Sigma^*$, $L(\emptyset)$ is $\emptyset$, and $L(r)$ is $\Sigma^*[r]\Sigma^*$.
- Set operators. Regular sets are closed under intersection, union, and complement, so if $L(E_1)$ and $L(E_2)$ are regular, then so are $L(E_1 \cap E_2)$, $L(E_1 \cup E_2)$, and $L(E_1 - E_2)$.
- Relational operators. Represent each algebra operator *op* by a finite state transducer (FST) mapping $L(E)$ to $L(\textit{op}\, E)$. The FSTs are shown in Figure 5.4. For example, the FST for *before* accepts input strings of the form $vwx[y]z$ and produces all possible output strings of the form $v[w]xyz$. Thus, the image of $L(E)$ under this FST is $L(\textit{before}\, E)$. By Theorem 3, if $L(E)$ is regular, then $L(\textit{op}\, E)$ is also regular for all relation operators *op*.

□

Corollary 4. *The inner language and outer language of any expression in $RST_{\mathcal{R}}$ are regular.*

The corollary follows from Theorem 5.

5.9 Context-Free Ground Terms ($RST_{\mathcal{CFL}}$)

We now come to the strongest variant of the region algebra under discussion in this chapter: algebra expressions with context-free grammar nonterminals as ground terms, $RST_{\mathcal{CFL}}$.

It should be noted that, unlike string literals and regular expressions, a grammar nonterminal implicitly constrains the context in which it can match. For example, consider the grammar

$$\begin{array}{rcl} S & \rightarrow & aS \,|\, Sb \,|\, A \\ A & \rightarrow & cde \end{array}$$

Although the nonterminal A matches a literal string cde, the algebra expression A is not equivalent to the literal expression cde. A can only match a region when the entire string can be derived from the grammar. Thus $L(A)$ includes $[cde]$ and $aaa[cde]bb$, but not $bb[cde]a$.

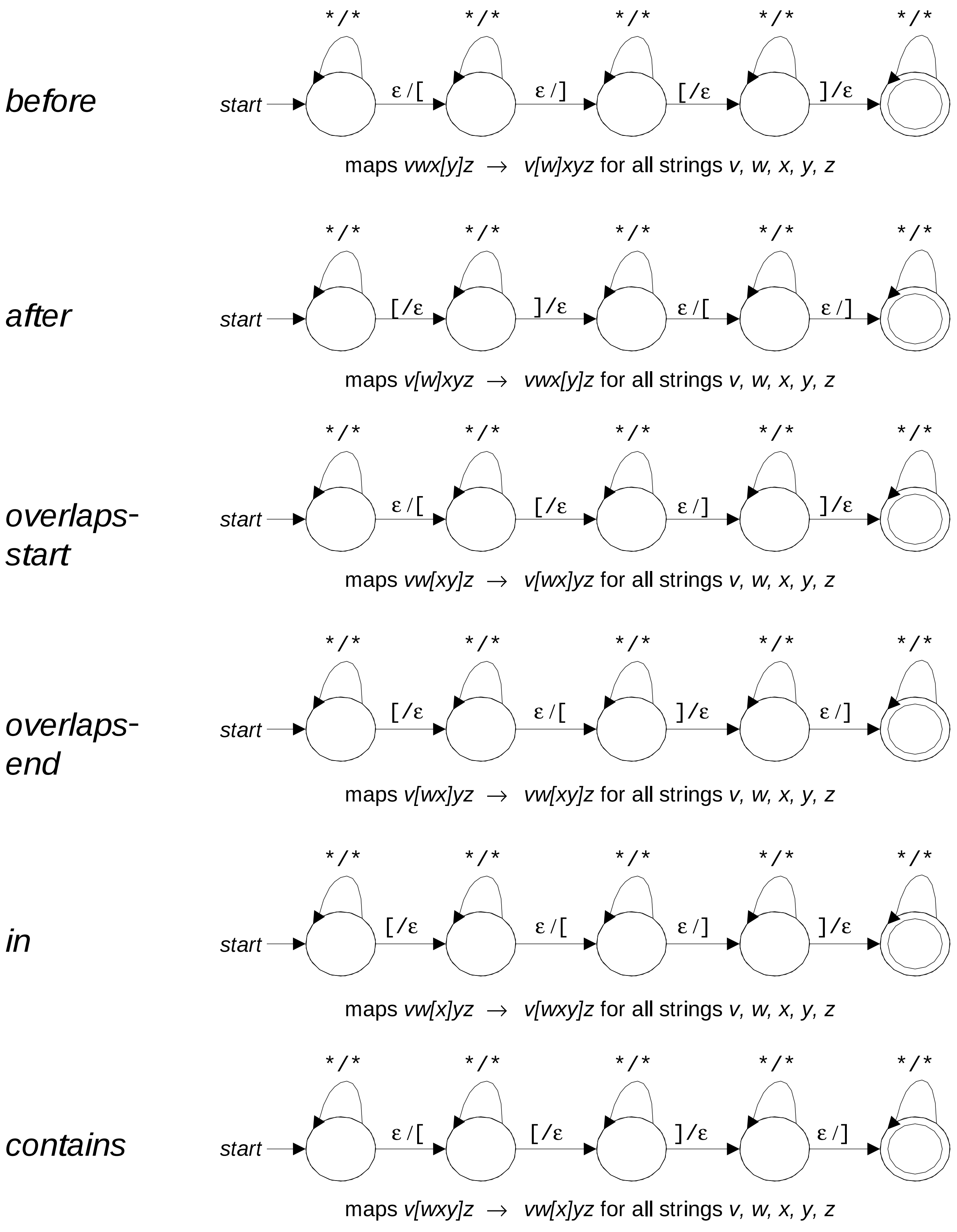

before
start
/
ε/[
ε/]
[/ε
]/ε
maps vwx[y]z → v[w]xyz for all strings v, w, x, y, z
after
start
/
[/ε
]/ε
ε/[
ε/]
maps v[w]xyz → vwx[y]z for all strings v, w, x, y, z
overlaps-start
start
/
ε/[
[/ε
ε/]
]/ε
maps vw[xy]z → v[wx]yz for all strings v, w, x, y, z
overlaps-end
start
/
[/ε
ε/[
]/ε
ε/]
maps v[wx]yz → vw[xy]z for all strings v, w, x, y, z
in
start
/
[/ε
ε/[
ε/]
]/ε
maps vw[x]yz → v[wxy]z for all strings v, w, x, y, z
contains
start
/
ε/[
[/ε
]/ε
ε/]
maps v[wxy]z → vw[x]yz for all strings v, w, x, y, z

Figure 5.4: Finite state transducers for the relational operators.

Nevertheless, since any literal string or regular expression can be represented by a context-free grammar G, literal and regular ground terms can be represented by a grammar nonterminal ground term (the start symbol of G), so $RST_{\mathcal{CFL}}$ contains all the expressions in $RST_{\mathcal{F}}$ and $RST_{\mathcal{R}}$.

Theorem 12. *If L is context-free, then L is the inner (and outer) language of some expression in $RST_{\mathcal{CFL}}$.*

Proof. Since L is context-free, there is some context-free grammar $G = (V, \Sigma, S, P)$ such that $L = L(G)$. Then the algebra expression S is the desired expression, since $L_0(S) = L(G)$ and $L_1(S) = L(G)$. □

Thus we have $\mathcal{CFL} \subset \mathcal{RST}_{\mathcal{CFL}}$. Unlike $RST_{\mathcal{R}}$, which could recognize no languages outside R, $RST_{\mathcal{CFL}}$ is stronger than $\mathcal{CFL}$.

Theorem 13. *There exists an expression in $RST_{\mathcal{CFL}}$ whose inner language and outer language are not context-free.*

Proof. Consider the two languages $L_1 = a^n b^n c^*$ and $L_2 = a^* b^n c^n$, recognized by the following grammars:

$$\begin{array}{rcl} S_1 & \rightarrow & X_1 C_1 \\ X_1 & \rightarrow & aX_1b \mid \epsilon \\ C_1 & \rightarrow & cC_1 \mid \epsilon \end{array}$$

$$\begin{array}{rcl} S_2 & \rightarrow & A_2 Y_2 \\ A_2 & \rightarrow & aA_1 \mid \epsilon \\ Y_1 & \rightarrow & bY_1c \mid \epsilon \end{array}$$

Although both L_1 and L_2 are context-free, their intersection $L_1 \cap L_2 = \{a^n b^n c^n\}$ is not context-free [HU79, Example 6.1]. Yet $L_1 \cap L_2$ is the inner (and outer) language of the region algebra expression $S_1 \cap S_2$. □

Since $RST_{\mathcal{CFL}}$ is strictly larger than $\mathcal{CFL}$, one might ask whether it is included in the context-sensitive languages, $\mathcal{CSL}$. The answer is in the affirmative:

Theorem 14. *If L is the region language of an expression in $RST_{\mathcal{CFL}}$, then L is context-sensitive.*

Proof. By structural induction on expressions in $RST_{\mathcal{CFL}}$.

- Ground terms. $L(\Omega)$ and $L(\emptyset)$ are clearly context-sensitive. So suppose the expression is a nonterminal A in some context-free grammar $G = (V, \Sigma, S, P)$. We will form a new context-free grammar $G' = (V_1 \cup V_2, \Sigma \cup \{[,]\}\ S_1, P')$ which puts square brackets around occurrences of A in its derivation, so that $L(G')$ is the region language of A. For every nonterminal N in the original grammar G, G' contains two nonterminals, N_1 and N_0. Intuitively, every N_1 will expand to a string containing square brackets, and every N_0 will expand to a string containing no square brackets. For every production $N \rightarrow \alpha$ in the original grammar, form all possible productions $N_1 \rightarrow \alpha_1$ by adding the subscript 1 to one nonterminal in α and 0 to all other

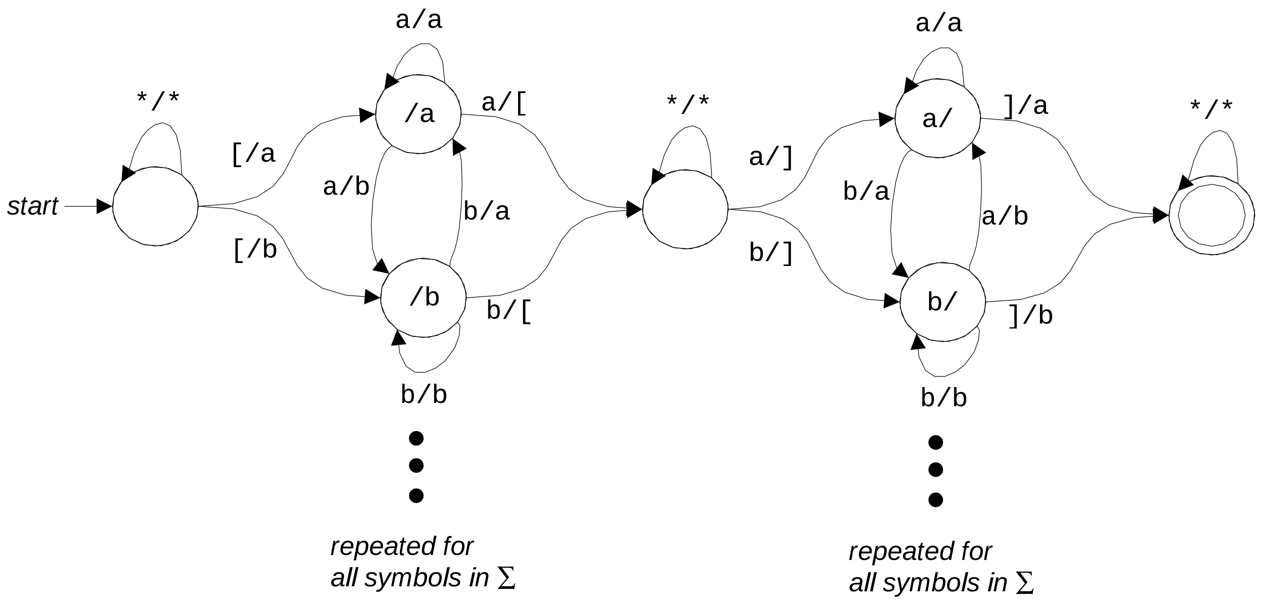


Figure 5.5: An ϵ-free FST for *in*.

nonterminals, and all productions $N_0 \to \alpha_0$ by adding the subscript 0 to all nonterminals in α. Finally, add the production $A_1 \to [A_0]$. The resulting grammar derives a string $x[y]z$ if and only if the original grammar derives xAz and thence xyz, so $L(G') = L(A)$. Since $L(G')$ is context-free, it is obviously context-sensitive.

- Set operators. Context-sensitive languages are closed under union, intersection, and complement. (Closure under union is well-known [HU79], but closure under intersection and complement is a relatively recent result [Imm88].) Thus if $L(E_1)$ and $L(E_2)$ are context-sensitive, then so are $L(E_1 \cap E_2)$, $L(E_1 \cup E_2)$, and $L(E_1 - E_2)$.

- Relational operators. Each relational operator can be represented by a finite state transducer, as shown in Figure 5.4. However, the FSTs shown in the figure incorporate ϵ transitions, and context-sensitive languages are only closed under ϵ-free FSTs (Theorem 4). However, each of the FSTs can be transformed into an equivalent ϵ-free FST by replicating states to remember symbols consumed but not yet emitted (or vice versa). At most two symbols must be stored at any time, so the set of states is still finite. For example, Figure 5.5 shows the ϵ-free FST for the *in* operator. As a result, if $L(E)$ is context-sensitive, then $L(\,op\,E)$ is also context-sensitive for any relational operator *op*.

$\square$

It remains an open question whether all context-sensitive languages can be recognized by $RST_{\mathcal{CFL}}$.

5.10 The *forall* Operator ($RSTA_{\mathcal{R}}$ and friends)

The arguments thus far have allowed only RST — algebra expressions containing relational operators, set operators, and concatenation. It remains to deal with the *forall* operator. The principal complication of *forall* is that it binds variables, so that the meaning of an expression E depends not only on E itself but on the bindings assigned to the free variables in E. As a result, the semantics of E will no longer be a set of strings, but instead a function mapping the bindings of its free variables to the set of strings it recognizes. This function takes a particular form, namely an *n-way regular relation.*

Before defining regular relations, let us illustrate the technique with a simple special case: expressions with only one free variable. To be precise, whenever *forall* $(\alpha : E_1) \,.\, E_2$ occurs in the expression, E_2 contains no occurrence of *forall*. Many of the uses of *forall* in Chapter 3 satisfy this restriction, particularly the adjacency and overlapping relational operators.

In *forall* $(\alpha : E_1) \,.\, E_2$, the expression E_2 contains a single free variable α that is bound to each region matching E_1. In terms of languages, the region language $L($*forall* $(\alpha : E_1) \,.\, E_2)$ is the set of strings generated by E_2 when α is bound to each string in the region language $L(E_1)$. Thus, the expression E_2 acts as a *transducer* which takes a string bound to α as input and produces a match to E_2 as output. When E_2 is a simple relational expression like *before* α, then it is easy to see that it is actually a *finite state transducer.* (Finite state transducers for the relational operators were given in Figure 5.4.) We will see that the same property extends to all operators in the algebra: for any expression E_2 with only one free variable, E_2 is a finite state transducer. The region language recognized by the *forall* expression, $L($*forall* $(\alpha : E_1) \,.\, E_2)$, is the language recognized by E_1 mapped through the finite state transducer for E_2. By Theorem 3, therefore, if $L(E_1)$ is regular, context-free, or context-sensitive, then $L($*forall* $(\alpha : E_1) \,.\, E_2)$ is likewise regular, context-free, or context-sensitive.

The following sections extend this technique to an arbitrary number of free variables by introducing n-way regular relations.

5.10.1 N-way Regular Relations

Regular relations, also called rational relations in the algebra literature, extend the concept of regularity to Cartesian products of languages. The development here follows Kaplan & Kay [KK94].

First we give some preliminary definitions. An *n-way string relation* is a set of n-tuples of strings over some alphabet Σ. Equivalently, an n-way string relation is a subset of the n-way Cartesian product $\Sigma^* \times \cdots \times \Sigma^*$. We write a tuple of strings as $< x_1, \ldots, x_n >$. For the n-tuple of empty strings, we write $\epsilon_n =< \epsilon, \ldots, \epsilon >$. String concatenation is extended over n-tuples as follows: if $x_1, \ldots x_n$ and $y_1, \ldots, y_n$ are strings, then

$$< x_1, \ldots, x_n >< y_1, \ldots, y_n >\equiv< x_1y_1, \ldots, x_ny_n >$$

The concatenation of two relations is the pairwise concatenation of their elements:

$$R_1R_2 \equiv \{< x_1y_1, \ldots, x_ny_n > \mid < x_1, \ldots, x_n >\in R_1 \text{ and } < y_1, \ldots, y_n >\in R_2\}$$

Exponentiation of an n-way relation is defined as expected:

$$\begin{array}{rcl} R^0 & \equiv & \{\epsilon_n\} \\ R^i & \equiv & R^{i-1}R \end{array}$$

The Kleene star R^* is defined as $\cup_{i=0}^{\infty} R^i$.

Regular relations are defined recursively as follows:

- $\emptyset$ and $\{< x_1, \cdots, x_n >\}$ for any $< x_1, \cdots, x_n > \in \Sigma^* \times \cdots \times \Sigma^*$ are regular relations;
- if R and R' are regular relations, then RR', $R \cup R'$, and R^* are regular relations;
- there are no other regular relations.

Every regular n-way relation is recognized by an n-tape finite state transducer, a generalization of the finite state transducer defined in Section 5.2. Every transition in an n-tape finite state transducer is labeled with n symbols, which may be either symbols in Σ or the empty string ϵ. Formally, a finite state transducer is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a set of states; Σ is the alphabet; δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \cdots \times (\Sigma \cup \{\epsilon\})$ to finite subsets of Q; q_0 is the starting state; and F is the set of accepting states.

It is easy to show that regular relations are closed under a number of operations:

- union $R \cup R'$;
- concatenation RR';
- Kleene star R^*;
- product $R \times R' = \{< x_1, \ldots, x_n, y_1, \ldots, y_m > \mid < x_1, \ldots, x_n > \in R$ and $< y_1, y_2, \ldots, y_m > \in R'\}$;
- join $R\,\mathit{join}\,R' = \{< x_1, \ldots, x_n, y_2, \ldots, y_m > \mid < x_1, \ldots, x_n > \in R$ and $< x_n, y_2, \ldots, y_m > \in R'\}$
- composition $R \circ R' = \{< x_1, \ldots, x_{n-1}, y_2, \ldots, y_m > \mid < x_1, \ldots, x_n > \in R$ and $< x_n, y_2, \ldots, y_m > \in R'\}$. Note that composition is like a join where the common component x_n is removed.

Regular relations are also closely related to regular languages. The projection $\pi_i(R) = \{x_i \mid < x_1, \ldots, x_n > \in R\}$. If L is a regular language, then the 1-relation $< L > = \{< x > \mid x \in L\}$ is a regular relation. The image of L under a regular relation R, written $R/_i L = \{< x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n > \mid < x_1, \ldots, x_n > \in R$ and $x_i \in L\}$, is also a regular relation. In particular, $R/_i \Sigma^*$ deletes the ith component of a n-way regular relation R to produce an $n-1$-way regular relation.

Regular relations are not, in general, closed under intersection and difference. For example, consider the two regular relations $R_1 = < \epsilon, b >^* < a, c >^*$ and $R_2 = < a, b >^* < \epsilon, c >^*$. The relation R_1 consists of tuples of the form $< a^n, b^* c^n >$, and relation R_2 is $< a^n, b^n c^* >$, so the intersection $R_1 \cap R_2$ is $< a^n, b^n c^n >$. This relation cannot be regular because its projection $b^n c^n$ is not regular.

Closure under intersection and difference is vital if we want to use regular relations for region algebra semantics. Fortunately, Kaplan and Kay show that a useful subclass of regular relations *is* closed under these operations. A *same-length* relation is a relation R such that for every $< x_1, \ldots, x_n >$ in R, $|x_1| = \cdots = |x_n|$. Same-length regular relations are closed under:

- intersection $R \cap R'$;
- difference $R - R'$;
- union $R \cup R'$;
- concatenation RR';
- Kleene star R^*;
- join $R\,\textit{join}\,R'$;
- composition $R \circ R'$;
- image with a regular language $R/_iL$.

Same-length regular relations are not closed under product $R \times R'$, however, since a tuple in R may be paired with a tuple in R' of a different length.

5.10.2 Algebra Semantics With *forall*

We are now ready to give semantics for region expressions with variables. Assume that all variables in an expression are distinct, so that every *forall* operator binds a unique variable. It is trivial to rewrite the expression with fresh variables to make this the case. Let V be the set of variables that can be used in a region expression, and $v \in V^*$ be a string of variables. For a string of bound variables v and a region expression E whose free variables can all be found in v, let $R_v(E)$ denote the $|v| + 1$-way relation defined as follows:

$$\begin{aligned} R_v(E) \equiv \{ \;& < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x[y]z > | \\ & [|x|, |xy|] \in E/xyz \text{ when each } v_i \text{ is bound to the region } [|x_i|, |x_iy_i|] \\ & \text{and } x_iy_iz_i = xyz \text{ for all } i\} \end{aligned}$$

For any tuple in $R_v(E)$, the first $|v|$ components represent the regions bound to variables, and the $|v| + 1$st component is the region matching the expression E.

Using this definition, we can write the region language $L(E)$ recognized by a region expression that contains variables. Assuming E has no free variables (but possibly bound variables), then

$$L(E) = \pi_1(R_\epsilon(E))$$

i.e., the projection of the 1-relation $R_v(E)$ where no variables are bound.

An important constraint in the definition of $R_v(E)$ is that for every tuple $< x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x[y]z >$ in $R_v(E)$, it must be the case that $x_1y_1z_1 = \cdots = x_ny_nz_n = xyz$. In other words, the n bound variables and the output must all refer to regions in the same string, xyz. A tuple that satisfies this constraint will be called *well-formed*. For convenience in later definitions, we let $\Delta \equiv \{x[y]z | xyz \in \Sigma^*\}$ be the set of all well-formed region strings, and

$$\Delta_n \equiv \{< x_1[y_1]z_1, \ldots, x_n[y_n]z_n > | x_iy_iz_i = w \text{ for some } w \text{ and all } i\}$$

be the set of all well-formed region tuples. It is always the case that $R_v(E) \subset \Delta_{|v|+1}$.

Lemma 2. Δ_n *is a same-length regular relation.*

Proof. By definition Δ_n is a same-length relation, because every string in a tuple has length $|w|+2$. The proof that Δ_n is regular goes by induction on n. The 1-way relation $\Delta_1 =< \Delta >$ is regular because the language Δ is regular. The 2-way relation Δ_2 is regular because it is the union of the six 2-way regular relations *before*, *after*, *in*, *contains*, *overlaps-start*, and *overlaps-end* described by the finite state transducers shown in Figure 5.4. (Claim 1 implies that the union of the six binary region relations gives all possible relations between regions.) For $n > 2$, $\Delta_n = \Delta_{n-1}$ *join* Δ_2, so by the induction hypothesis and closure of same-length regular relations under join, Δ_n is regular. □

The next section will show that $R_v(E)$ is also a same-length regular relation for expressions E in $RSTA_{\mathcal{R}}$.

The semantics of all region algebra operators, including *forall*, can be written in terms of $R_v(E)$. We omit concatenation, since it was shown in Chapter 3 that concatenation can be defined in terms of *forall*.

Ground Terms

The ground terms include literal strings w, regular expressions r, and context-free grammar nonterminals A. A ground term ignores the bound variables and returns the region language $L(E)$. However, the entire relation $R_v(E)$ must still satisfy the constraint that all strings in a tuple are identical (modulo the positions of the square brackets). Since Δ_n expresses this constraint, we can write $R_v(E)$ for ground terms E as follows:

$$\begin{aligned}
R_v(w) &= \Delta_{|v|+1}\,\textit{join}\, < L(w) > \\
R_v(r) &= \Delta_{|v|+1}\,\textit{join}\, < L(r) > \\
R_v(A) &= \Delta_{|v|+1}\,\textit{join}\, < L(A) >
\end{aligned}$$

Set Operators

The set operators are straightforward:

$$\begin{aligned}
R_v(E_1 \cap E_2) &= R_v(E_1) \cap R_v(E_2) \\
R_v(E_1 \cup E_2) &= R_v(E_1) \cup R_v(E_2) \\
R_v(E_1 - E_2) &= R_v(E_1) - R_v(E_2) \\
R_v(\Omega) &= \Delta_{|v|+1} \\
R_v(\emptyset) &= \emptyset
\end{aligned}$$

Relational Operators (R)

The relational operators are represented as 2-way finite state transducers in Figure 5.4, which are equivalent to 2-way string relations. Let $R(op)$ be the 2-way relation for *op*. Then we can write each $R_v(\,op\,E)$ as a composition as follows:

$$
\begin{aligned}
R_v(\textit{before}\, E) &= R_v(E) \circ R(\textit{before}\,) \\
R_v(\textit{after}\, E) &= R_v(E) \circ R(\textit{after}\,) \\
R_v(\textit{in}\, E) &= R_v(E) \circ R(\textit{in}\,) \\
R_v(\textit{contains}\, E) &= R_v(E) \circ R(\textit{contains}\,) \\
R_v(\textit{overlaps-start}\, E) &= R_v(E) \circ R(\textit{overlaps-start}\,) \\
R_v(\textit{overlaps-end}\, E) &= R_v(E) \circ R(\textit{overlaps-end}\,)
\end{aligned}
$$

Forall Operator

Finally we consider the *forall* operator. The expression *forall* $(\alpha : E_1) \,.\, E_2$ evaluates E_1 and binds a new variable α which is added to the variable subscript v when evaluating E_2. In detail, R_v for this expression is

$$
\begin{aligned}
R_v(\textit{forall}\,(\alpha : E_1) \,.\, E_2) = \{ \; & < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x[y]z > | \\
& < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha > \in R_v(E_1) \\
& \text{and } < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha, x[y]z > \in R_{v\alpha}(E_2)\}
\end{aligned}
$$

We will want to show that this is a same-length regular relation, so let us rewrite it as a combination of $R_v(E_1)$ and $R_{v\alpha}(E_2)$ using operators under which same-length regular relations are closed. Starting with the well-formed $n+1$-way relation $R_v(E_1)$, we extend it to a same-length $n+2$-way relation by joining it with Δ_2:

$$
\begin{aligned}
R_v(E_1)\, \textit{join}\, \Delta_2 = \{ \; & < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha, x[y]z > \\
& < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha > \in R_v(E_1) \\
& \text{and } x_i y_i z_i = xyz\}
\end{aligned}
$$

This expression represents the bound variables $v\alpha$ allowed by E_1. To find the result of E_2 given those bindings, we intersect with $R_{v\alpha}(E_2)$ to get

$$
\begin{aligned}
(R_v(E_1)\, \textit{join}\, \Delta_2) \cap R_{v\alpha}(E_2) = \{ \; & < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha, x[y]z > | \\
& < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha > \in R_v(E_1) \\
& \text{and } < x_1[y_1]z_1, \ldots, x_n[y_n]z_n, x_\alpha[y_\alpha]z_\alpha, x[y]z > \in R_{v\alpha}(E_2)\}
\end{aligned}
$$

Finally, we apply $/_{|v|+1}\Delta$ to delete the α component of the relation so that the result is a $n+1$-way relation omitting the bound variable α, as required. Thus

$$
R_v(\textit{forall}\,(\alpha : E_1) \,.\, E_2) = ((R_v(E_1)\, \textit{join}\, \Delta_2) \cap R_{v\alpha}(E_2))/_{|v|+1}\Delta
$$

5.10.3 $RSTA_{\mathcal{R}} = R$

Now we are ready to show that $R_v(E)$ is a same-length regular relation, which will lead immediately to the result that $RSTA_{\mathcal{R}}$ recognizes exactly the class of regular languages.

Theorem 15. *$R_v(E)$ is a same-length regular relation for all expressions E in $RSTA_{\mathcal{R}}$.*

Proof. By structural induction on E.

- Ground terms. For a regular expression r, $R_v(r) = \Delta_{|v|+1}$ *join* $< L(r) >$ is a same-length regular relation because Δ_n is same-length regular, $< L(r) >$ is regular and trivially same-length, and same-length regular relations are closed under join. Likewise, $R_v(\Omega) = \Delta_{|v|+1}$ and $R_v(\emptyset) = \emptyset$ are same-length regular.

- Set operators. Same-length regular relations are closed under intersection, union, and difference.

- Relational operators. For each operator *op*, $R(\textit{op})$ is a same-length regular relation, and same-length regular relations are closed under composition, so $R_v(\textit{op}\, E) = R_v(E) \circ R(\textit{op})$ is same-length regular.

- Forall operator. Since $R_v(\textit{forall}\,(\alpha : E_1)\,.\,E_2) = ((R_v(E_1)\,\textit{join}\,\Delta_2) \cap R_{v\alpha}(E_2))/_{|v|+1}\Delta$, and same-length regular relations are closed under join, intersection, and deletion of a component, $R_v(\textit{forall}\,(\alpha : E_1)\,.\,E_2)$ is same-length regular.

□

Corollary 5. *If L is the region language for some expression E in $RSTA_{\mathcal{R}}$, then L is regular.*

Proof. $L = L(E)$ is a projection of the regular relation $R_\epsilon(E)$, so L is regular. □

5.10.4 Open Questions: $RSTA_\emptyset$, $RSTA_{\mathcal{F}}$, $RSTA_{\mathcal{CFL}}$

We have seen that adding the *forall* operator to the region algebra over regular expressions does not affect its power. Like $RST_{\mathcal{R}}$, $RSTA_{\mathcal{R}}$ recognizes the class of regular languages, no more, no less. What about the other algebra specializations: no ground terms, finite ground terms, or context-free ground terms? Does *forall* affect their power?

Adding *forall* to $RST_\emptyset$ clearly expands the set of region languages that can be recognized — not a hard job considering that $RST_\emptyset$ can only recognize $L(\Omega)$ and $\emptyset$. For example, *forall* enables the definition of operators like *zero-length*, which matches only the zero-length regions in a string, and *nth*$_n A$, which matches the nth region matching A. Combining these operators produces the expression

$$\textit{ends nth}_{n+1} zero - length$$

which matches only regions containing exactly $n + 1$ zero-length regions, i.e., regions of length n. Since $RSTA_\emptyset$ offers no way to test the characters of the string, however, it seems plausible that $RSTA_\emptyset$ recognizes a class of region languages that depends only on the location of the square brackets and not on the symbols in the string. As a conjecture, $RSTA_\emptyset$ might recognize the set of region languages L such that for any substitution function $h : \Sigma \to \Sigma$ that preserves [and], $h(L) = L$. Characterizing this language class precisely and proving that $RSTA_\emptyset$ recognizes it is left as future work.

Whether $RSTA_{\mathcal{F}}$ recognizes only noncounting languages, like $RST_{\mathcal{F}}$ does, is also an open question. The reader may have noticed that the proof techniques for $RST_{\mathcal{F}}$ were substantially

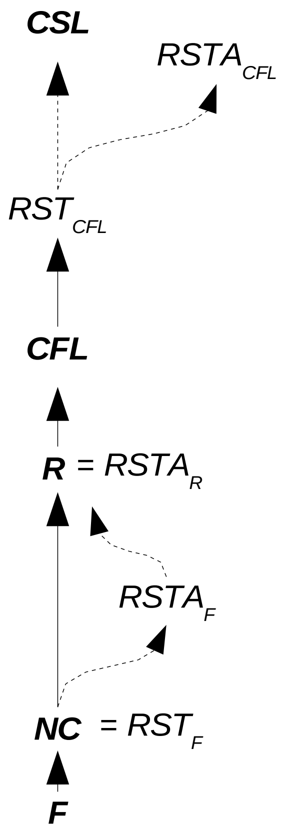


Figure 5.6: Summary of relationships among language classes.

different from $RST_{\mathcal{R}}$ and $RST_{\mathcal{CFL}}$, primarily because noncounting languages are not closed under mapping through an arbitrary finite state transducer. The regular languages are the smallest language class that is closed under FST mapping [HU79, Lemma 11.1]. Since $NC \subset R$, we could not use finite state transducers to show that $RST_{\mathcal{F}}$ recognizes NC. As a result, the regular relation machinery developed for showing that $RSTA_{\mathcal{R}}$ recognizes R will not help to decide whether $RSTA_{\mathcal{F}}$ recognizes NC.

Finally, we consider $RSTA_{\mathcal{CFL}}$. We know that $\mathcal{CFL}$ is a strict subset of $RSTA_{\mathcal{CFL}}$, but is $RSTA_{\mathcal{CFL}}$ a subset of $\mathcal{CSL}$? I believe it is, but resolving the question would require developing some additional theoretical machinery which is left as future work.

5.11 Summary

The important relationships between the region algebra variants discussed in this chapter and the the language classes of the Chomsky language hierarchy are summarized in Figure 5.6. In the figure, solid arrows denote proper inclusion between language classes, e.g., $\mathcal{R} \subset \mathcal{CFL}$. Dashed arrows denote inclusion which is not known to be proper. For example, it is trivial that $RST_{\mathcal{F}} \subseteq RSTA_{\mathcal{F}}$, but it is not known whether $RST_{\mathcal{F}} = RSTA_{\mathcal{F}}$. The dashed arrows represent open problems, as do missing lines, such as the relationship between $\mathcal{CSL}$ and $RSTA_{\mathcal{CFL}}$.

Chapter 6

Text Constraints

The pattern language in LAPIS is called *text constraints* (TC). The TC language has the region algebra at its core, in much the same way that the regular expression pattern languages in Perl and awk are fundamentally based on regular expressions, but with additional operators to make the language expressive enough for practical use. Since the semantics of region algebra operators have already been presented in Chapter 3, the current chapter focuses largely on the syntax of the language.

TC incorporates some usability guidelines into its design, in the spirit of Natural Programming [Mye98]. TC avoids the keyword *and*, which is dangerously ambiguous for reasons discussed in the next section. TC also uses a simple, universal rule for operator precedence and associativity, and avoids most parentheses by deducing expression structure from indentation. Although indentation has been used to specify block structure in other languages (notably Python and Haskell), TC is the first language that uses indentation for structuring infix expressions.

The usability of TC was tested in a small user study, and the results of the user study were fed back into language design. The description of TC presented in this chapter includes the changes that resulted from the user study. These changes and the user observations that motivated them are discussed when the corresponding language features are presented in this chapter. However, since the user study evaluated not only TC but also other features of the LAPIS user interface in which TC is embedded, a complete discussion of the study itself is postponed until Chapter 7.

6.1 Goals

There were three primary goals in the design of TC.

- **Expressiveness.** The language should be able to combine and constrain lightweight structure using all the power of the region algebra.
- **Readability.** Novice users should be able to read and comprehend expressions in the language without difficulty, so that an intelligent system that infers patterns from examples can display the patterns as one form of feedback.
- **Low error rate.** Novice users should be able to write TC expressions without making the kinds of syntactic and logical errors that are common in other programming languages, particularly *and*/*or* confusion and operator precedence problems.

These goals were motivated by perceived weaknesses in the prevailing text pattern languages: regular expressions and grammars. Both regular expressions and grammars fail the expressiveness test, because they cannot express intersection or negation of patterns. Both also are notoriously difficult to read, partly because of their heavy use of punctuation operators (*, +, ?, [], $, ^, |), and partly because the languages' low expressiveness forces many patterns to be long and complicated. Finally, like most programming languages, regular expressions force users to memorize precedence and associativity rules, resulting in errors. Although recent research has improved the readability and writability of regular expressions by using a visual notation [Bla01, JB97], this work did nothing for expressiveness.

From a survey of the novice programming literature, Pane and Myers distilled a list of usability guidelines for designers of novice programming systems [PM96]. Of the 29 guidelines in 8 categories, the following were the most useful for designing TC:

- **Beware of misleading appearances.** The language should not encourage creation of expressions that can be easily misinterpreted. A classic example of this problem is a program whose indentation is inconsistent with its block structure. TC avoids this problem by treating indentation as grammatically significant, so that indentation affects both the user's interpretation and the system's interpretation of an expression.

- **Avoid subtle distinctions in syntax.** The language should avoid subtle syntactic distinctions that have drastically different semantics. In Perl, for example, the single quote (') and double quote (") look very similar but have considerably different meanings. In almost all languages, 123 and "123" are completely different types. In TC, all quote marks have the same semantics, and 123 and "123" both refer to the literal string pattern 123, since TC has no need to distinguish numeric constants from string constants.

- **Consistency in notation.** The language should be self-consistent, uniform, and free from unnecessary exceptions.

- **Naturalness of the programming language.** Draw on natural language where appropriate to make the language more familiar or mnemonic, but beware of encouraging users to form overly high expectations of the system's ability to understand freeform natural language. Like other end-user programming languages, most notably HyperTalk [App02], TC uses natural keywords and a syntax that mimics natural language phrase structure for many common patterns.

- **Viscosity.** Viscosity measures how much programmer effort is required to make a small change to a program. Visual programming languages typically have high viscosity, since large parts of the program may need to be shuffled around to make room for the change. Old versions of BASIC had high viscosity, since lines may need to be renumbered to make room for a new line of code. Similarly, when indentation is used to represent expression structure, making a small change to an expression may require changing the indentation of a large block of code. In order to make TC less viscous in this respect, the TC editing environment in LAPIS automatically adjusts the indentation of subexpressions as the user types.

- **Use signaling to highlight important information.** Secondary signaling, such as syntax coloring or indentation, give perceptual cues to the meaning of an expression. The TC editor in LAPIS follows this principle, not only using indentation but also drawing lines showing a pattern's expression structure.

TC is not a general-purpose programming language, so its design does not raise as many challenges that the design of a general-purpose programming language would. Yet there are particular syntactic issues, known to be troublesome from general-purpose programming language design, that TC must face squarely:

- **Boolean operators.** Observations of novice programmers and database users have shown that users tend to confuse *and* with *or*, probably because these words are used ambiguously in natural language [GDCG90, PM00]. The more troublesome of the two operators seems to be *and*, since it can be used in natural language to mean not only Boolean conjunction ("I want candy bars that are chewy and sweet") but also disjunction ("I want candy bars from Hershey and Cadbury"). One study of SQL expressions written by novice users found that it is more common to use *and* incorrectly in place of *or* than vice versa [GDCG90]. TC incorporates these observations into its design by omitting the *and* keyword entirely. Boolean conjunction is represented implicitly by predicate application. Since expert users expect to be able to use *and* and *or*, however, TC also provides an expert mode in which *and* has its conventional meaning.

- **Operator precedence, associativity, and parentheses.** Any language with infix operators faces the problem of precedence and associativity. For example, C has 17 levels of operator precedence, and some of its operators are left-associative while others are right-associative [KR88]. Operator precedence and associativity are well-known problem areas for both novice and expert programmers. In fact, programming style books often recommend redundant parentheses when the precedence or associativity is not completely obvious [KP99]. Unfortunately, parentheses are not an ideal solution either, because a number of studies have shown that novice users tend to misinterpret or ignore parentheses [GDCG90, PM00]. TC avoids precedence and associativity confusion by establishing a common rule for all operators, and avoids most parenthesis problems by structuring expressions with indentation.

6.2 Language Description

This section describes the TC pattern language. An alphabetical reference to all the operators in the language may be found in Appendix B.

6.2.1 Basic Lexical Rules

TC is case-insensitive, so keywords and identifiers may be written in uppercase, lowercase, or mixed capitalization without affecting their interpretation. Case-insensitivity also applies to literal patterns and regular expressions by default, so the literal pattern `"apple"` matches not only `apple`, but also `Apple` and `APPLE`. Literals and regular expressions may be made case-sensitive using the `case-sensitive` keyword, described in more detail in Section 6.2.8.

Line breaks are significant in TC. A line break ends one or more subexpressions, and the indentation of the subsequent line indicates which parent expression is resumed. Expression indentation is described in more detail in Section 6.2.16.

Comments are started by # and terminated by a line break. Comments can begin anywhere in a line. For example:

```
# This is a comment
"apple"  # This is also a comment
"#this is not a comment because it’s quoted"
```

6.2.2 Basic Syntactic Rules

All TC expressions are drawn from the same syntactic class, hereafter called *expr*. In the sections that follow, TC syntax is defined by syntactic productions of the form

```
expr ::= production
```

Taken together, all the productions make up a grammar for TC, although an ambiguous one. The parse of a TC expression is determined unambiguously by a combination of precedence rules, indentation, and explicit parentheses. A full discussion of these rules is postponed until Section 6.2.16, after the TC operators have been presented. For the time being, the examples will use explicit parentheses to clarify how an expression should be parsed. In practice, however, most users would write these examples with indentation instead of explicit parentheses.

Either parentheses or curly braces can be used to bracket an expression explicitly:

```
expr ::= (expr)
expr ::= {expr}
```

TC makes no semantic distinction between the two kinds of delimiters. In other languages, curly braces are used to bracket statements, and parentheses to bracket expressions. Since some TC expressions are naturally regarded as statements (e.g., pattern definitions) and others as expressions (patterns themselves), it seems natural to allow both kinds of delimiters but not enforce any difference between them. Parentheses and curly braces are not completely interchangeable, however: a left parenthesis must be terminated by a right parenthesis, not a right curly brace.

Despite the usability problems of explicit parentheses, they are included in TC for three reasons: (1) to allow TC expressions to be written as arguments on a program’s command line, where linebreaks and indentation would be difficult to specify; and (2) to allow long lines to be continued; and (3) to match the expectations of expert users.

6.2.3 Pattern Identifiers

Any pattern may be named by an identifier. An identifier is a sequence of any characters except whitespace, quotes, parentheses, curly braces, =, or #, as long as it does not match a TC keyword, literal, or regular expression:

```
id ::= [^ \t\n’"(){}=#]+
```

Note that the space of identifiers permitted by TC is larger than most other languages, which typically allow only sequences of alphanumeric characters or underscores in identifiers. As a result, TC identifiers can include punctuation such as `<`, `>`, `[`, and `]`, a property which is put to good use by the HTML parser described in the next section.

A TC expression may reference a pattern by its identifier:

```
expr ::= id
```

The assignments between identifiers and patterns are stored in a global library. The patterns in this library are not just TC expressions. In general, a pattern is a Java object that implements the `Pattern` interface:

```
interface Pattern {
    RegionSet match (Document doc);
}
```

Given some document, the `match` method returns the set of regions that match the pattern in the document.

All compiled TC expressions implement the `Pattern` interface, but nothing prevents writing an arbitrary Java object that implements `Pattern`. This allows external parsers to be integrated into TC, as described in the next section.

6.2.4 Parsers

A parser is a Java class that defines a collection of related patterns for some kind of text structure, like HTML or Java syntax. These patterns are generally *not* TC expressions. Instead, the patterns are small Java objects that not only represent structure found by the parser (such as grammar nonterminals) but also serve as entry points into the parser. This section describes the architecture of parsers in more detail.

A parser must implement the `Parser` interface:

```
interface Parser {
    void bind (PatternLibrary lib);
}
```

A parser's `bind` method is called only once, when the parser is first loaded by LAPIS. The `bind` method takes the global pattern library as its only argument, and assigns each of the parser's pattern objects to the appropriate name in the library. The `bind` method essentially performs dynamic linking that makes the parser's identifiers available for use in TC expressions.

Notice that the `Parser` interface has no method for causing a parser to scan a document. Instead, parsing is triggered by calling the `match` method of one of the parser's pattern objects. The parsers currently in LAPIS use a call to `match` as an opportunity to find the matches to *all* of the parser's patterns in a single pass over the document. The region sets found by this parsing pass are then stored in a cache, so that a future application of any of the parser's patterns to the same document can simply look up the answer in the cache.

This design mediates between the traditional approach to parsing, which uses a left-to-right scan through a document, and the region-set approach, which uses the region algebra to manipulate the entire document at once. Parsers generated by parser-generators like lex, yacc, and JavaCC are easy to integrate into this system, and so are hand-coded scanners. At the same time, the pattern library is kept simple. An identifier is always bound to a single pattern, and a pattern applied to a document always produces a single region set. The TC evaluator, and the user, need not be aware of how this region set was produced: whether by a TC expression using region algebra operations or by a traditional parser using a left-to-right scan.

LAPIS includes three built-in parsers. A complete list of the identifiers bound by each parser can be found in Appendix A, but here is a brief overview:

- The HTML parser is a hand-coded finite state machine, which makes it extremely tolerant of illegal HTML. The HTML parser binds several kinds of identifiers into the pattern library. *Tags* are represented by suggestive identifiers like `<p>`, `</p>`, `<a>`, `</a>`, and `<img>`. (Note that there is nothing special about the use of punctuation like <, /, and > in these identifiers. The punctuation marks are not TC operators; they are simply part of the identifier.) *Elements* match complete HTML elements from start tag to end tag (if any). The identifier for an element is the tag name in square brackets, such as `[p]`, `[a]`, and `[img]`. Finally, *attributes* match a name-value attribute in a tag. Examples include `href-attr` or `width-attr`.

- The Java parser is generated from a grammar by the JavaCC parser-generator [Jav00]. The grammar is based on the Java grammar included with JavaCC. The original grammar produced an abstract syntax tree from Java source. In order to make the generated parser into a LAPIS parser, the syntax tree node classes are augmented with region information (start and end offset), and whenever a node of a certain type (such as `Statement` or `Expression`) is reduced by the parser, the node's region is added to the region set for that type. Adapting the JavaCC grammar to LAPIS required about 100 lines of code, most of which is generic code that would adapt any JavaCC-generated parser. No changes to the grammar productions were required.

- The Character parser is a hand-coded lexical analyzer that detects runs of character classes, such as `Letters`, `Digits`, and `Whitespace`. These patterns could be represented by independent regular expressions instead, but the Character parser is more efficient because it makes only a single pass over the document.

6.2.5 Pattern Definitions

A TC expression may be assigned to a pattern identifier by the `is` operator:

expr ::= *id* **is** *expr*

This expression assigns *expr* to *identifier* in the global pattern library. Like all TC expressions, an assignment expression also has a value — it returns the region set matching *expr* in the current document. This behavior is convenient when defining patterns interactively, because it causes LAPIS to highlight the region set matched by the pattern definition.

If another pattern is already assigned to the identifier in the library, then the new assignment replaces it. Before the new assignment is made, however, all identifiers in the new definition are first *bound*, i.e., replaced with their current assignments in the library. For example, consider the following definitions:

```
    Fruit is "apple" or "orange"
    Fruit is Fruit or "banana" or "pear"
```

When the first definition is evaluated, the pattern `"apple" or "orange"` is assigned to the identifier `Fruit`. The second definition redefines `Fruit`, but first binds the original definition for `Fruit` so that `Fruit` effectively represents the pattern `("apple" or "orange") or "banana" or "pear"`.

Binding identifiers at definition time allows library patterns to be constrained or generalized without access to their original definitions and without danger of infinite regress. However, early binding has a tradeoff — any definitions added to the library between the original definition of `Fruit` and its redefinition continue to use the original definition of `Fruit`. For example, suppose another definition is evaluated between the two `Fruit` definitions:

```
    Fruit is "apple" or "orange"
    Produce is Vegetable or Fruit
    Fruit is Fruit or "banana" or "pear"
```

Since the reference to `Fruit` in `Produce` is bound at definition time, it continues to be bound to just `"apple" or "orange"` even after `Fruit` itself is extended to include other possibilities. Thus, with early binding, the meaning of a definition is dependent on when it was added to the library.

An alternative strategy is late binding: the definition for an identifier is not resolved until the expression is actually evaluated. With late binding, `Produce` would always use the latest definition of `Fruit`. However, late binding can fall victim to infinite regress when implemented naively. For example:

```
    Fruit is Fruit or "banana" or "pear"
```

With naive late binding, this expression evaluates to

```
    (Fruit or "banana" or "pear") or "banana" or "pear"
```

and so on. Late binding needs a way to prevent or break the recursion.

Probably the simplest solution is to forbid recursive definitions. Many macro languages adopt this strategy, including the C preprocessor (which has the rule that a macro can be expanded at most once in any given macro expansion). The no-recursion requirement could be checked at definition time by searching for cycles in the dependency graph, so that the system can raise an error as soon as a definition creates a cycle. A disadvantage of forbidding recursive definitions is that users would be unable to redefine an identifier like `Fruit` in terms of itself. Every change to `Fruit` would require the user to edit the original definition, which might be complex and intimidating.

In order to allow revisions of definitions, one could permit self-referential expressions (using early binding for the self-reference, but late binding for all other references) and forbid mutually recursive definitions. However, it is often useful to write mutually constraining definitions:

```
Paragraph is [p] containing Sentence
Sentence is (from CapitalizedWord to ".") in Paragraph
```

These definitions indicate that a Paragraph must contain a Sentence and a Sentence must lie in a Paragraph. With early binding, the first definition would generate an error, because `Sentence` is not yet bound. With late binding, the definitions would be accepted, but evaluating them would cause an infinite regress. In this particular case, the mutual recursion can be removed with an auxiliary definition that serves as a base case:

```
Paragraph0 is [p]
Sentence is (from CapitalizedWord to ".") in Paragraph0
Paragraph is Paragraph0 containing Sentence
```

LAPIS uses this technique to define `Paragraph` and `Sentence` in its built-in library. Obviously, however, this solution can only represent a bounded depth of recursion.

Another tradeoff between early binding and late binding is related to the LAPIS implementation. When a pattern like `Produce` is matched against a document, the resulting region set is stored in a cache associated with the document, so that future searches for `Produce` can simply look up the answer. The cache entry is invalidated when the document changes or when the definition of `Produce` changes. With late binding, however, the meaning of `Produce` may change whenever a change is made to any of the definitions it depends on, such as `Fruit`. When `Fruit` is redefined under a late binding regime, it must invalidate the cache entries for all the patterns that depend on it.

To summarize the tradeoffs, early binding makes definitions order-dependent, and late binding must check for dependency cycles and invalidate cache entries of dependent definitions. The LAPIS implementation currently uses early binding, which is easy to implement. In hindsight, for the reasons given above, a combination of early binding (for self-referential definitions like `Fruit`) and late binding (for other references) might have been preferable.

6.2.6 Visible and Hidden Identifiers

Many of the patterns in the LAPIS library are defined by TC expressions. When defining patterns for reuse in a library, however, it is important to be able to control the public interface exposed by your patterns — which identifiers are accessible to clients of the library, and which should be considered internal to the implementation. TC supports this distinction with a typographic convention: any identifier starting with an at-sign (@) is a *hidden identifier*.

For example, the definitions for `Paragraph` and `Sentence` given in the previous section would be improved if `Paragraph0` were a hidden identifier:

```
@Paragraph is [p]
Sentence is (from CapitalizedWord to ".") in @Paragraph
Paragraph is @Paragraph containing Sentence
```

These definitions produce only two public, visible identifiers, `Paragraph` and `Sentence`, but `@Paragraph` is kept hidden from the user, so that it can be renamed or removed as the definitions of `Paragraph` and `Sentence` evolve.

Hidden identifiers are treated differently from visible identifiers in several ways:

- Hidden identifiers do not appear in the LAPIS pattern library browser (Figure 6.1). A mode that shows all identifiers, both visible and hidden, would be useful for debugging purposes, but LAPIS does not currently provide it.

- Hidden identifiers are not automatically imported from other namespaces (see the next section).

- Hidden identifiers are not used for inference (Chapter 9) or outlier finding (Chapter 10).

Hidden identifiers are not protected, however. If a user knows that `@Paragraph` is a hidden identifier defined by the `Paragraph`/`Sentence` definitions, then the user can write a pattern referring to `@Paragraph` just like a visible identifier. Hidden identifiers do not enforce information hiding; they merely encourage it. However, only an advanced user of TC — in particular, one who wants to write a modular system of patterns to be installed in the library — should need to know about or use hidden identifiers.

Hidden identifiers can also be overwritten by new definitions, just like visible identifiers. A potential disadvantage of hidden identifiers — collisions between hidden names — is alleviated by the namespace mechanism described in the next section. Hidden identifiers are generally used only inside a namespace.

6.2.7 Namespaces

Every identifier in TC belongs to some *namespace*. Namespaces are hierarchical pathnames, with the components separated by dots. The root namespace is denoted by a single dot, “." For example, `.Business.Address.ZipCode` refers to the identifier `ZipCode` in the namespace `.Business.Address`. An identifier may simultaneously denote a pattern and a namespace; for example, `.Characters.Whitespace` is an identifier denoting runs of whitespace, but it is also the namespace for `.Characters.Whitespace.Spaces` and `.Characters.Whitespace.Tabs`, which denote runs of particular kinds of whitespace.

Namespaces have two purposes in TC. First, they make the pattern library easier to browse by subdividing it into hierarchical categories. Figure 6.1 shows a screenshot of the LAPIS pattern library, which organizes the identifiers in the library by namespace. Second, namespaces reduce the risk of collisions between identifiers and permit modularity in parsers and definitions. For example, a Java parser and a C++ parser can both use identifiers like `Statement`, `Type`, and `Expression` as long as those identifiers are placed in different namespaces.

Many languages use some kind of namespace or package mechanism, including C++, Tcl, Perl, and Java. In all these languages, the user must explicitly import other namespaces in order to access their identifiers. TC takes a different approach. In TC, *all* namespaces are imported automatically. Lookup rules give preference to identifiers in the current namespace, and ambiguous identifiers must be disambiguated by an explicit namespace prefix. These rules are described below.

Every TC expression is evaluated relative to some namespace, called the *current namespace*. By default, the current namespace is the root namespace (dot). The current namespace may be changed by the `prefix` operator:

```
expr ::= prefix identifier expr
```

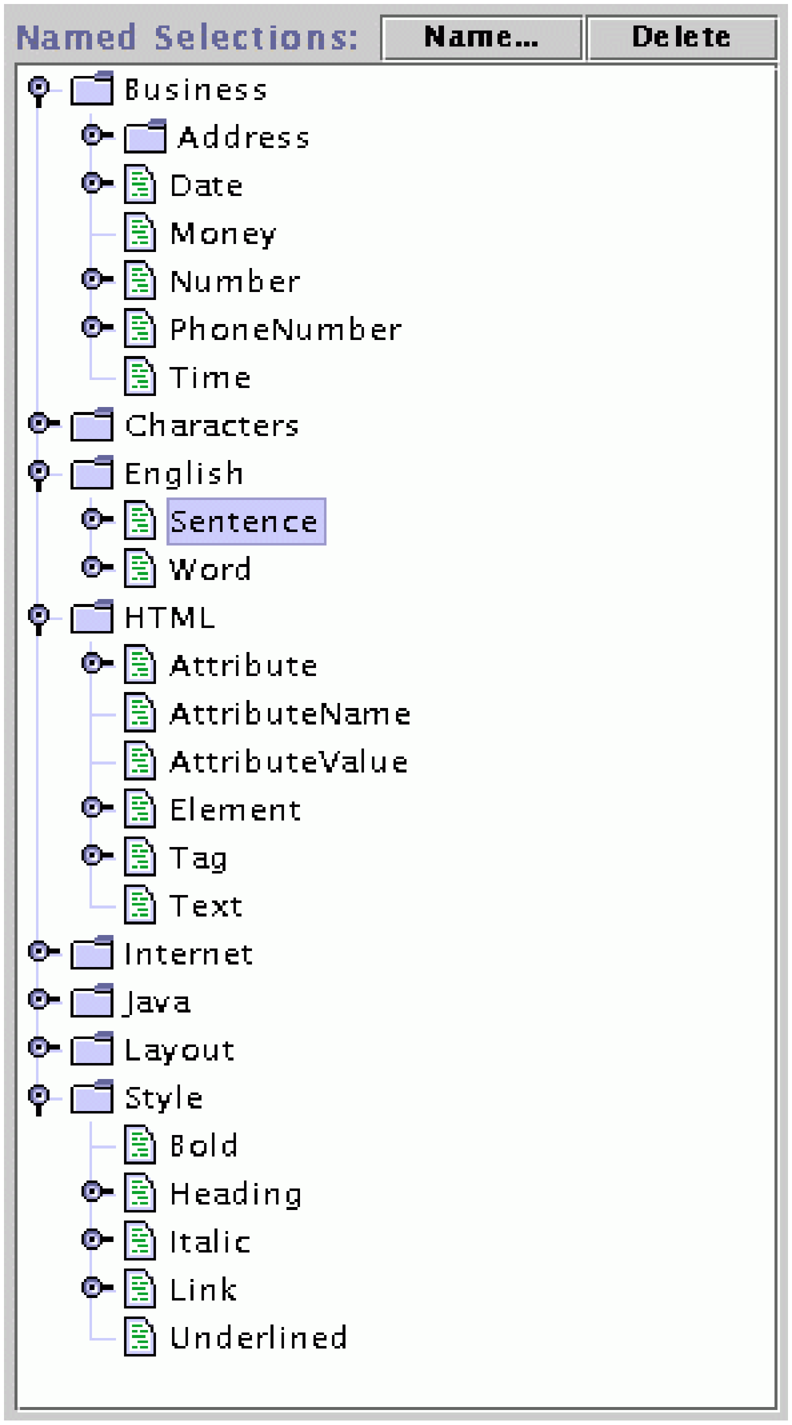


Figure 6.1: The LAPIS pattern library shows the namespace hierarchy.

This expression changes the current namespace to *identifier* for the scope of *expr*. For example,

```
# current namespace is .
prefix Grocery {
   # current namespace is .Grocery
   prefix Produce {
       # current namespace is .Grocery.Produce
   }
}
```

Any identifier, whether denoting a namespace or a pattern, is either absolute or relative. An absolute identifier starts with a period, e.g., `.Grocery.Produce.Fruit`. An absolute identifier describes a complete path from the root namespace. A relative identifier lacks a starting period; examples include `Grocery.Produce.Fruit`, `Produce.Fruit`, or just `Fruit`. A relative identifier must be converted into an absolute identifier by interpreting it relative to the current namespace. However, the interpretation of relative identifiers depends on whether the identifier is being *defined* (by a `prefix` or `is` expression) or *used* (referenced in a pattern).

When a relative identifier is used in a definition, the relative identifier is simply appended to the current namespace. For example, all of the following definitions define the identifier `.Grocery.Produce.Fruit`:

```
prefix Grocery {
   prefix Produce {
       Fruit is "apple" or "orange"
   }
}

prefix Grocery.Produce {
    Fruit is "apple" or "orange"
}

Grocery.Produce.Fruit is "apple" or "orange"

prefix SomeOtherNamespace {
    prefix .Grocery.Produce {
        Fruit is "apple" or "orange"
    }
}
```

The last example in particular shows that namespaces are not protected from each other. Any namespace can define identifiers in any other namespace using an absolute identifier.

When a relative identifier is referenced in a pattern, TC searches for the identifier in the pattern library using the following procedure. Assume the current namespace is *namespace* (an absolute identifier) and the referenced identifier is *identifier* (either relative or absolute).

1. If *identifier* is absolute, then look up *identifier* in the library. If *identifier* is found, return it. Otherwise report that *identifier* is not found.

2. Look up *namespace.identifier*. If it is found, return it; otherwise, continue searching.

3. Look up all identifiers named **.identifier,* where * is a namespace path of any length starting from the root, possibly empty. If there is exactly one match, return it. If there is more than one match, report an error to the user listing all the identifiers matching **.identifier* and suggesting an unambiguous identifier for each one.

4. Report that *identifier* is not found.

The name resolution algorithm is designed so that most users can ignore the existence of namespaces entirely. If an identifier occurs uniquely (in only one namespace in the library), then the identifier can always be used directly, with no need for a namespace qualifier or explicit import. All the identifiers in the built-in LAPIS library are unique, so the user can refer to `PhoneNumber` or `URL` or `Whitespace` without knowing or caring which namespaces these identifiers are found in. If the user defines a pattern with the same name as an existing identifier, then the user's pattern takes precedence, since the user's definition is stored in the current namespace (the root namespace, by default) and the current namespace always takes precedence over other namespaces. Thus a user can write

```
Statement is Sentence ending "."
```

and proceed to use `Statement` to refer to this definition, blissfully unaware that the Java parser also defines an identifier `Statement` with completely different semantics in the `Java` namespace. The shadowed meaning of `Statement` is still accessible with `Java.Statement`, and any expressions evaluated in the `Java` namespace continue to use the Java version of `Statement`.

When two namespaces define the same identifier, and neither one is the current namespace, the user must add a namespace qualifier to disambiguate them. As an interesting side-effect of step 3 in the name resolution algorithm, however, the qualified name need not be absolute. It is sufficient to use *any suffix* of the namespace path that uniquely describes the desired identifier. For example, suppose the library contains two identifiers named `State`: `.Business.Address.State`, and `.Programming.Verilog.State`. Then the former identifier can be written uniquely as `Address.State`, and the latter as `Verilog.State`. As a result, a qualified name need not be longer than necessary.

The main drawback of TC's namespace mechanism is that adding new identifiers to the library may cause existing patterns to break. For example, suppose a C++ parser is added to the library, so that the identifiers `Statement`, `Expression`, and `Type` occur not only in the `Java` namespace but also in the `C++` namespace. Then existing patterns that refer to `Statement` will subsequently fail with an error message, because `Statement` is ambiguous without a qualifier. Namespaces in other languages also suffer from this problem, but to a lesser degree. For example, a Java program that imports all the classes in the `java.io` package may become uncompilable if a class with a conflicting name is added to the `java.io` package, but not if a class is added to some `foo.bar.baz` package that the program never refers to.

TC patterns can be made robust to this kind of change by using absolute identifiers for all references outside the current namespace. To make this more convenient, hidden local identifiers can be defined as aliases for the foreign identifiers. For example:

```
prefix MyNamespace {
    @stmt is .Java.Statement
    @word is .English.Word
    IfStatement is @stmt starting @word = "if"
}
```

The hidden identifiers `@stmt` and `@word` have the effect of explicitly importing the foreign identifiers `.Java.Statement` and `.English.Word` into the `MyNamespace` namespace. This technique is essentially identical to the explicit-import-with-aliasing feature found in Modula-3 and Python. The only missing piece is an automatic check that *all* foreign references are absolute, in order to avoid accidental foreign references due to misspellings or omitted definitions. This check could be made by simply disabling step 3 in the name resolution algorithm when the user requests it with a special directive. TC does not yet provide this directive, however.

6.2.8 Literals

A string in quotes returns the set of all regions that literally match the string:

```
expr ::= "literal"
expr ::= ’literal’
```

Either single quotes or double quotes may be used, with the same effect.

A literal pattern may return an overlapping region set. For example, when the pattern "aa” is matched against the string `aaaa`, the resulting region set contains three regions: `[aa]aa`, `a[aa]a`, and `aa[aa]`. A literal pattern can only return an overlapping region set if some strict prefix of the pattern is also a suffix of the pattern.

By default, literal matching ignores alphabetic case. Literal matching can be made case-sensitive using the `case-sensitive` operator:

```
expr ::= case-sensitive expr
```

This operator has the effect of making matching case-sensitive for the entire scope of *expr*, which may be a single literal:

```
case-sensitive "apple"
```

or may be an entire system of definitions:

```
case-sensitive {
    keyword is "while" or "do" or "if" or "for" ...
}
```

The complementary operator `not case-sensitive` can be used to revert to case-insensitive matching within the scope of a `case-sensitive` operator:

```
expr ::= not case-sensitive expr
```

No metacharacters or escape codes are recognized within the literal string. In particular, the backslash character just means backslash. Metacharacters are troublesome because they impose a memory load on users, and using backslash as an escape code causes trouble on platforms that use backslash for other purposes, such as separating components of a pathname. TC provides other ways to match characters that require escape codes in other systems:

- Single quotes or double quotes can be matched by using the other quote character to delimit the string. For example, the pattern `"Bob's your uncle"` matches a single quote by surrounding it with double quotes, and `'He said "hello" to me'` does the reverse. If a literal match contains both single quotes and double quotes, then it can be split up into multiple concatenated patterns using `then`:

  ```
  'He said "' then "Bob's your uncle" then '" to me'
  ```

- The library patterns Linebreak and Tab match \n and \t.

Advanced users who want to use escape codes to match nonprintable characters can fall back to regular expressions, described below, in which all the usual escape codes are available.

6.2.9 Inferring Quotes

When TC is used for interactive pattern matching, quoting literal patterns can be a hassle. Observations from the TC user study bear this out: novice TC users frequently forget to put quotes around literal patterns. Quoting has also been identified as a key usability problem by other researchers [Bru97]. It seems likely that novice users are tempted to omit quotes by prior experience with Find commands in other text editors, since these commands generally accept *only* a literal pattern, with no need for quoting.

TC addresses the problem of omitted quotes by automatically inferring quotes around single tokens when the token is neither a keyword nor a defined identifier. Quote inference is implemented by adding a step to the name resolution algorithm from Section 6.2.7:

1. If *identifier* is absolute, then look up *identifier* in the library. If *identifier* is found, return it. Otherwise report that *identifier* is not found.

2. Look up *namespace.identifier*. If it is found, return it; otherwise, continue searching.

3. Look up all identifiers named **.identifier* (where * is a namespace path of any length). If there is exactly one match, return it. If there is more than one match, report an error to the user listing all the identifiers matching **.identifier* and suggesting an unambiguous identifier for each one.

4. **Search for *identifier* as a literal pattern in the current document. If at least one match is found, then treat this pattern as the literal pattern "*identifier*".**

5. Otherwise report that *identifier* is not found.

Since TC identifiers are drawn from a very large lexical class — only whitespace, quotes, parentheses, curly braces, # and = cannot appear in an identifier — this technique can automatically quote many common kinds of constants, such as:

```
55.3
$200
247-3940
5/18/02
www.cmu.edu
http://www.yahoo.com/
```

Automatic quote inference has a number of tradeoffs. First, the technique only applies to identifier tokens. TC cannot automatically quote multiword phrases, nor can it quote a word that happens to be a keyword, like `in`, `or`, or `before`. Although it is tempting to try automatic multiword quoting whenever a syntax error occurs, the number of ways to "correct" an expression by adding quotes explodes exponentially with the number of tokens in the expression. If a multiword expression contains no reserved words or special characters, however, it may be reasonable to quote the entire expression automatically. For example, if the user wrote

```
Gettysburg Address
```

then the system would interpret it as the quoted literal "Gettysburg Address". This approach is used by Bruckman's MOOSE system [Bru97]. One problem with automatic multiword quoting, however, is that system error messages become more complex when neither the original pattern nor the quoted pattern succeeds. The gist of the error message would be:

```
No matches found, either because Gettysburg Ad-
dress isn't a properly formed pattern or because "Gettys-
burg Address" isn't found in the document.
```

Automatic quoting of a single-word pattern like `Gettysburg` doesn't face this problem, because an error message like

```
No matches found, because Gettys-
burg isn't found in the library or the document.
```

adequately describes the situation. For this reason, TC does not currently do automatic multiword quoting.

Another problem with automatic single-word quoting is that a user accustomed to omitting quotes from literals may write a quoteless pattern that accidentally refers to a identifier defined in the library. For example, suppose the user executes

```
line containing state
```

intending it to be interpreted as `line containing "state"`, but the `state` identifier actually resolves to the pattern `.Business.Address.State` matching states of the US. This can lead to a disturbing situation where the user can *see* regions that match the intended pattern, but the actual pattern is matching something different, with no error message to suggest what might have gone wrong.

Similarly, automatic single-word quoting can make references to undefined identifiers appear legal when the identifier happens to occur as a literal in the document. This is less of a problem for interactive patterns than it is for pattern definitions written for reuse in the pattern library. A future version of TC will have a directive that disables automatic quoting, so that expert users can write patterns for reuse that are checked so that they only use explicit quoting.

6.2.10 Regular Expressions

A regular expression surrounded by slashes returns the set of regions matching the regular expression:

```
expr ::= /regexp/
```

For example,

```
/[0-9]+-[0-9]+-[0-9]+/
```

matches three groups of digits separated by dashes, such as 333-555-12203. Like literal patterns, regular expression patterns are case-insensitive by default. A regular expression may be made case-sensitive using the `case-sensitive` operator.

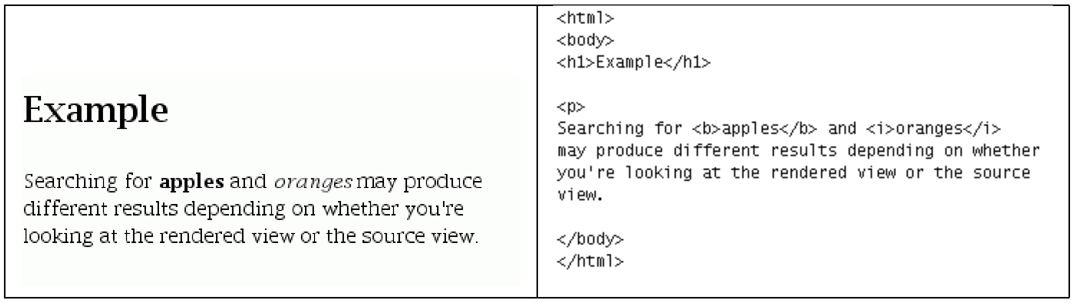
LAPIS uses an off-the-shelf regular expression library to match regular expressions, so the syntax and semantics of a regular expression depends how the library interprets it. LAPIS currently uses the Jakarta-regexp library for Java [Jak99]. The operators recognized by this library are shown in Figure 6.2.

The regular expression library does not return *all* regions that match a regular expression. It returns a *nonoverlapping* region set of matches found in a left-to-right scan of the string. For example, when the expression `/aa/` is matched against the string `aaaa`, the resulting region set includes only the two regions `[aa]aa` and `aa[aa]`. Furthermore, when the regular expression contains a greedy repetition operator *, +, or ?, the expression matches as many occurrences of the repetition as possible, so `/a.*a/` matches *all* of the string `abbaaba`. Conversely, the nongreedy repetition operators *?, +?, and ?? match as few occurrences as possible. Matching `/a.*?a/` against the string `abbaaba` returns two regions, `[abba]aba` and `abba[aba]`.

This behavior is inherent in the regular expression library, and conforms to the expectations of users familiar with regular expressions in other systems. One unfortunate consequence, however, is that the literal pattern `"aa"` is not equivalent to the regular expression `/aa/`, since the former can return an overlapping region set but the latter cannot. One might fix this particular inconsistency by restarting the regular expression matcher after every match — i.e., after the matcher returns a region $[s, e]$, restart it from position $s + 1$. This change would also affect how other regular expressions were interpreted: for example, `/a*/` matched against `aaaa` would produce four regions, `[aaaa]`, `a[aaa]`, `aa[aa]`, `aaa[a]`, and `aaaa[]`, which is different from its interpretation in

Expression	Matches
character	*character* itself
.	any character
[abc]	character class: any of a, b, or c
[a-zA-Z]	character range a-z or A-Z
[^abc]	any character except a, b, or c
E*	zero or more occurrences of E
E+	one or more occurrences of E
E?	zero or one occurrences of E
(E)	grouping
E|F	either E or F
EF	E followed by F
\c	quotes a metacharacter *c*
\n, \r, \t	newline, carriage return, tab
\0nnn, \xhh	an ASCII character in octal (0nnn) or hexadecimal (xhh)
\\\\	backslash
\1, \2, \3, etc.	the same as the *n*th parenthesized subexpression
^	the start of a line
$	the end of a line
\w, \s, \d	word, space, or digit character
\W, \S, \D	non-word, non-space, or non-digit
\b, \B	word boundary or non-word boundary
[:alpha:], [:alnum:], [:digit:], etc.	predefined character classes
E*?, E+?, E??	nongreedy closures: matches as *few* occurrences of E as possible
E{n}, E{n,}, E{n,m}	exactly n, at least n, or between n and m occurrences of E

Figure 6.2: Regular expression operators supported by LAPIS.



Example

Searching for **apples** and *oranges* may produce different results depending on whether you're looking at the rendered view or the source view.

```
<html>
<body>
<h1>Example</h1>

<p>
Searching for <b>apples</b> and <i>oranges</i>
may produce different results depending on whether
you're looking at the rendered view or the source
view.

</body>
</html>
```

(a) (b)

Figure 6.3: Different views of a simple HTML page: (a) rendered; (b) source.

other systems. The dilemma is between internal consistency with TC literal patterns, and external consistency with regular expressions in other systems. Since regular expressions are included in TC largely for the sake of expert users to transfer their experience from systems like Perl and awk, the dilemma is resolved in this case in favor of external consistency.

For future work, it would be interesting to implement a regular expression library that can produce *all* the regions matching a regular expression. Such a library would be more consistent with the behavior of other TC operators, but considerable care would be needed to make it run as efficiently as existing regular expression libraries.

6.2.11 Matching Against HTML

LAPIS can display web pages as well as text files. By default, web pages are displayed in a *rendered view* in which the HTML markup is interpreted as formatting instructions (Figure 6.3(a)). A web page may also be viewed in a *source view* in which the HTML markup is visible (Figure 6.3(b)). A literal or regular expression pattern can specify which of these views it should match against.

By default, a literal or regular expression is matched against the current view. When the rendered view is displayed as in Figure 6.3(a), the literal pattern `"apples and oranges"` matches the text shown. When the source view is showing as in Figure 6.3(b), the literal pattern must be `"<b>apples</b> and <i>oranges</i>"` to match the equivalent place in the text.

A TC expression can force matching against a particular view using the `view source` or `view rendered` operators:

```
expr ::= view source expr
expr ::= view rendered expr
```

Any literal or regular expression patterns in the scope of *expr* is matched against the specified view, unless overridden by another `view source` or `view rendered` operator.

The current view has no effect on patterns defined by parsers (Section 6.2.4). A parser may choose to look to scan the source view, the rendered view, or both when it parses a document. For example, the HTML parser parses the source view, while the Character parser parses the rendered view.

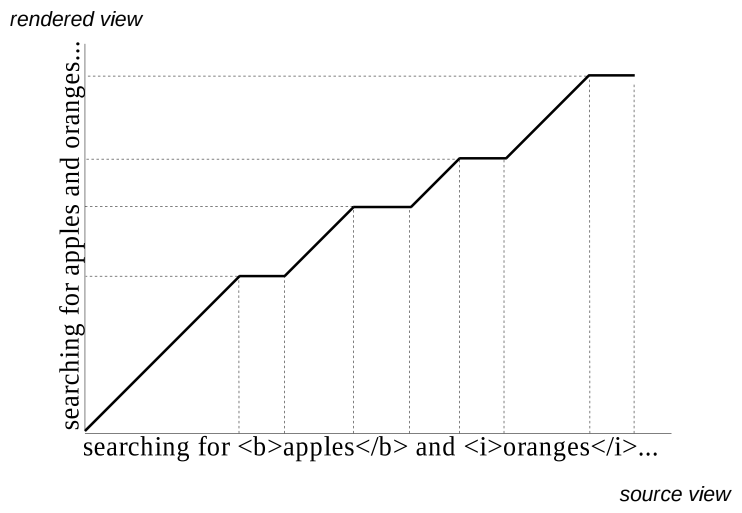


Figure 6.4: Coordinate map mapping from source view to rendered view for part of an HTML document.

6.2.12 Coordinate Maps

Each view has a different *coordinate system*, a mapping of integers to character positions. The coordinate systems of the two views are related to each other by a *coordinate map*, a binary relation that equates an offset in one view to an equivalent offset in the other view. A simple coordinate map that maps between source and rendered views is shown in Figure 6.4. A coordinate map is not necessarily a function, but all coordinate maps are monotonically increasing in the sense that if $[a, b]$ and $[c, d]$ are related by the map, then either $a \leq c$ and $b \leq d$ or $a \geq c$ and $b \geq d$. The inverse of a coordinate map from source view to rendered view is the coordinate map from rendered view to source view. Although the domain and range of a coordinate map are technically nonnegative integers, it is convenient to extend the map over the reals (as shown in Figure 6.4) and regard it as a piecewise linear, monotonically increasing curve. This perspective allows a coordinate map to be represented compactly by the endpoints of the linear pieces in sorted order. A coordinate can be transformed through a coordinate map in $O(\log N)$ time by finding the endpoints above and below it with binary search and then interpolating.

Coordinate maps are used to transform pattern matches found in one view into the other view. By convention, all region sets returned by TC expressions are relative to the source-view coordinate system, so that TC operators that combine region sets need not do any coordinate transformations. When a literal or regular expression pattern matches against the rendered view, it transforms the resulting region set into source-view coordinates before returning it.

Coordinate maps are also used to highlight the result of a pattern match in the rendered view. Since the pattern matcher returns a region set in source-view coordinates, LAPIS must transform the region set to rendered-view coordinates before it can highlight the regions.

Coordinate maps are also used to translate region sets between different versions of a document. This technique is used by simultaneous editing to update cached region sets as the user edits the document (Chapter 9).

TC Syntax	Algebra Operator	Section
expr? **in** *expr*	*in*	3.5.2
expr? **contains** *expr*	*contains*	3.5.2
expr? **equals** *expr*	*equals* $_W$	3.6.13
expr? **just before** *expr*	*just-before* $_W$	3.6.13
expr? **just after** *expr*	*just-after* $_W$	3.6.13
expr? **starts** *expr*	*starts* $_W$	3.6.13
expr? **ends** *expr*	*ends* $_W$	3.6.13
expr? **anywhere before** *expr*	*before*	3.5.2
expr? **anywhere after** *expr*	*after*	3.5.2
expr? **overlaps** *expr*	*overlaps*	3.6.3
expr? **overlaps start of** *expr*	*overlaps-start*	3.5.2
expr? **overlaps end of** *expr*	*overlaps-end*	3.5.2
expr **then** *expr*	*then* $_W$	3.6.13
from *expr* **to** *expr*	*fromto*	3.6.8
balanced from *expr* **to** *expr*	*balances*	3.6.8
start of *expr*	*start-of*	3.6.1
end of *expr*	*end-of*	3.6.1
expr **trim** *expr*	*trim*	3.6.14
*n***th** *expr*	*nth*	3.6.5
*n***th** *expr* (**before|after|in**) *expr*	$nth - op$	3.6.5
flatten *expr*	*flatten*	3.6.12
melt *expr*	*melt*	3.6.12
nonzero *expr*	*nonzero*	3.6.4

Figure 6.5: TC pattern-matching operators with their equivalent region algebra operators from Chapter 3. Several of the relational operators can be either unary or binary, as indicated by the optional first operand `expr?`.

6.2.13 Pattern Matching Operators

At the heart of the TC language lie the region algebra operators (Chapter 3). TC supports the six fundamental relations of the region algebra (*in, contains, before, after, overlaps-start,* and *overlaps-end*), although several of them are renamed for usability reasons explained below. In addition to these fundamental operators, TC provides most of the derived operators whose semantics were defined in Chapter 3. The complete set of pattern-matching operators is shown in Table 6.5.

The names of the operators were chosen with some care. Some of these design decisions were made by following the usability principles listed in Section 6.1. Others resulted from observing the errors made by users in the TC user study, described in Section 7.6. Some of the more interesting design decisions are discussed below.

- **One official keyword, several unofficial synonyms.** Table 6.5 lists only one keyword for each operator, but TC actually accepts a number of synonyms, shown in Table 6.6. For example, *starts* can also be written as *starting*, *starts with*, *starting with*, *beginning with*, or *at start of*. All patterns presented by the system — in tutorial materials, online help, or self-disclosure — only use the official keyword, to avoid giving the impression that *starts* and

Operator	Synonyms
`in`	inside, of
`contains`	containing, containg *[sic]*
`equals`	equal to, equalling, equaling, =
`just before`	directly before, right before, jbef
`just after`	directly after, right after, jaft
`anywhere before`	abef
`anywhere after`	aaft
`starts`	starts with, starting, starting with, at start of,
	begins, begins with, beginning, beginning with, at beginning of
`ends`	ends with, ending, ending with, at end of,
	finishes, finishing, finishes with, finishing with
`nth`	*n*st, *n*nd, *n*rd, *n*d, first, second, third, ..., tenth
`trim`	trimming, trimming off
`not`	but not

Figure 6.6: Synonyms for TC operators.

begins might mean different things (following the rule Avoid Misleading Appearances). But by accepting synonyms as well, the system becomes more tolerant of a user's faulty memory. Synonyms are also useful as abbreviations of commonly-used but verbose keywords. For example, an experienced TC user can just use *jbef* instead of writing out *just before*. The tradeoffs of accepting synonyms are twofold. First, synonyms consume more reserved words, reducing the space of identifiers available to the programmer. Second, synonyms must be chosen judiciously. A word or phrase that might be a synonym for more than one operator should not be accepted for any of them. For example, *followed by* is a possible synonym for *just before*, but it might also mean *anywhere before*, *then*, or *from-to*, so TC doesn't accept this synonym for any of them.

- **Containing vs. contains**. Originally, the official keyword for the *contains* operator was *containing*. This decision was justified by the fact that it makes TC expressions read more like grammatically correct English, which, it was hoped, would make the expressions both easier to generate and easier to comprehend. Finding a "line containing apple" sounds better than finding a "line contains apple". However, users tended to misspell *containing* as *containg* — a natural mistake, since the keyword is quite long and repeats *in* twice. After several users made this mistake, the official keyword was changed to *contains*, which is shorter and less apt to be misspelled, and TC was modified to accept both *containing* and *containg* as synonyms for *contains*. Subsequent users made no spelling mistakes with *contains*. Accepting common misspellings of keywords is not a new idea, of course. For example, AMPL [FGK93] accepts *sovle* as a synonym of *solve*.

- **Concatenation.** In other pattern languages, notably regular expressions and grammars, concatenation is expressed implicitly by juxtaposing expressions, as in *expr1 expr2 expr3*. This approach was considered but discarded in favor of an explicit concatenation keyword, on the grounds that an explicit keyword would contribute more to TC's readability. The choice of keyword, *then*, was motivated by a study [PRM01] which found that nonprogrammers describing a programming task in natural language often used *then* to describe sequencing.

- **Starts vs. at start of/starting with.** An early version of TC provided two operators for the same-start relation: *at start of* and *starts with*. The *at start of* operator was equivalent to the algebra operator *starts-in*, that is, it matched regions that both started at the same place and lay completely inside another region. The *starts with* operator, on the other hand, was equivalent to the algebra operator *starts-contains*, which matched regions that started at the same place and completely contained another region. This choice of operators was later reconsidered in light of usability guidelines (Avoid Subtle Distinctions in Syntax), since it seemed likely that users would confuse *at start of* and *starts with*. Fortunately, for many patterns, the direction of containment is implicit. For example, in the pattern *Word starting with "s"*, it is unnecessary to have the extra constraint that the Word must completely contain "s", since that is implicit in the definition of Word. For common uses, then, the two operators can be replaced by a single operator *starts* that requires only that two regions have the same start point, saying nothing about which region contains the other. The user now has only one operator to remember, and no subtle distinctions between keywords: *at start of* and *starts with* are now synonyms for the *starts* operator. In the occasional pattern where the containment relation must be specified, it can be made explicit by combining *starts* with *in* or *contains*. The *ends* operator followed a parallel evolution.

- **Before vs. anywhere before**. Although *before* is one of the six fundamental region relations, it is not as useful in practice as the more specialized operators that are derived from it, particularly *just before*. *Before* is a much weaker constraint than *just-before*. An expression like *before R* effectively splits the string into two parts, and only the last region in *R* plays any role in this split — most of the structure described by *R* is completely irrelevant. Thus *before* is useful only for splitting the entire string around a single landmark, which is not often done. The expression *just-before R*, on the other hand, is a constraint that depends on the position of every region in *R*, and it may “split” the string into as many pieces as there are regions in *R*.

  Since *just before* is far more useful in practice than *before*, there is no particular need for the *before* operator to have such a short name, especially since it is prone to confusion with *just before* (Avoid Misleading Appearances). As a result, the operator was renamed to *anywhere before*, which clearly disambiguates it from *just before*. If the user writes a pattern simply using *before*, the system generates an error message asking the user to choose between *just before* and *anywhere before*. Similarly, the *after* operator is renamed to *anywhere after*.

6.2.14 Ignoring Background

Several relational operators test whether two regions have coincident endpoints: *just before*, *just after*, *starting*, *ending*, *equals*, and *then*. As discussed in Section 3.6.13, it is useful to weaken these tests so that endpoints are considered equivalent as long as they are only separated by irrelevant regions, called the background.

All the adjacency-testing operators in TC have an implicit background parameter, specified as a region set. Any characters covered by the background region set are considered irrelevant.

File Type	Default background
Plain text	Whitespace, punctuation
HTML	Whitespace, punctuation, tags
Java	Whitespace, comments

Figure 6.7: Default background for different file types.

Since the background is specified by a region set, instead of merely a set of character classes like whitespace and punctuation, the background can include rich high-level structure like HTML markup tags or Java comments.

The default background depends on the type of document being matched against. Table 6.7 shows the default background used by LAPIS for the different document types it recognizes. This background can be changed for the scope of an expression using the *ignoring* operator:

```
expr ::= expr ignoring background
```

This operator sets the background to the result of evaluating *background*, for all the adjacency operators in the lexical scope of *expr*. Background can be turned off, so that adjacency tests require strict adjacency, using `expr ignoring nothing`. The current background is available as the pattern identifier `Background`, so that an `ignoring` expression can incrementally add or remove regions from the current background.

In TC, the background is lexically scoped, not dynamically scoped, so changing the background with `ignore` does not affect any patterns that are referenced through pattern identifiers. This is a side-effect of early binding of pattern identifiers, and has the same advantages and disadvantages as early binding (Section 6.2.5).

In other text-processing systems, the effect of background is achieved by tokenizing the text, discarding irrelevant characters like whitespace, punctuation, or comments. Subsequent processing operates only on the resulting stream of tokens. In these systems, the background has already been discarded by the time pattern-matching or other processing occurs, so individual patterns cannot specify which characters should be ignored. In TC, on the other hand, the background is explicit, and it can be varied from pattern to pattern.

6.2.15 Boolean Operators

Three TC operators remain to be discussed: the operators for union, intersection, and difference. Previous research has shown these operators to be error-prone (Section 6.1), so some effort was devoted to choosing a syntactic representation that minimize the level of ambiguity or confusion.

Probably the most troublesome word is *and*. Although *and* is the most natural English word for describing Boolean conjunction, appearing as a keyword in nearly every high-level programming language since Fortran, it is also highly ambiguous, having many other meanings in English that novice programmers may inadvertently attribute to the keyword. *And* is not a satisfactory mnemonic because it suggests too many possible meanings. Unfortunately, no simple alternative is any better: words like *with, also, plus, likewise* do not seem to reduce the ambiguity.

TC avoids the ambiguity of *and* for novice users by providing no intersection operator by default. Instead, intersection is represented by relational operators. For example, the algebra expression

$$\textit{Word} \cap \textit{starts}\ \text{"c"} \cap \textit{ends}\ \text{"e"}$$

is represented by the TC expression

```
(Word starting "c") ending "e"
```

In general, any algebra expression of the form $A \cap$ *op* B or $($ *op* $B) \cap A$ can be represented without intersection by the TC expression `A` *`op`* `B`.

If neither operand of the intersection is a unary relational operator, then we introduce one — in particular, the *equals* operator. For example, the algebra expression

$$\textit{Word} \cap \text{"apple"}$$

might be converted first to the algebra expression

$$\textit{Word} \cap \textit{equals}\ \text{"apple"}$$

which then becomes the TC expression

```
Word equals "apple"
```

In general, any expression of the form $A \cap B$ can be represented by the TC expression `A equals B`. One might argue that this means that `equals` is simply taking the place of `and` as the intersection keyword, but this is not entirely accurate. First, `equals` is used only for intersecting "noun-like" expressions, i.e. expressions that are not predicates. Naively replacing every intersection with `equals`, even predicate expressions, would result in unreadable expressions like

```
Word equals (starting "c") equals (ending "e")
```

Second, `equals` provides background-sensitive intersection, which treats two regions as equal as long as their corresponding endpoints differ only by irrelevant background characters. For intersecting noun-like expressions, this is often precisely what is desired. For example, to find lines that contain a URL by itself, one would write

```
Line equals URL
```

which would match correctly even if the `Line` regions included the linebreak and the `URL` regions omitted it, as long as the current background includes whitespace. If the background is set to nothing, then `equals` behaves like ordinary intersection.

Despite the dangers of `and`, several users in the user study noticed its absence and asked for it. To meet these users' expectations, and for consistency with other query languages, TC also provides the `and` keyword:

```
    expr ::= expr and expr
```

By default, however, using `and` in a pattern generates a warning that explains its ambiguity and suggests alternative ways to express each of `and`'s possible meanings. This warning can be turned off by the user, if desired.

Union is specified by the `or` operator:

expr ::= *expr* **or** *expr*

which can be written more verbosely as follows:

expr ::= **either** *expr* **or** *expr*

The `either` keyword is useful for grouping the expression using indentation, explained in more detail in the next section.

Finally, set difference is specified by `not`:

expr ::= *expr* **not** *expr*

In practice, the second operand of `not` is usually a relational operator:

```
EmailAddress not ending "cmu.edu"
```

but it may also be a noun-like expression:

```
Fruit not "orange"
```

As mentioned in Table 6.6, `but not` may be used as a synonym for `not`.

6.2.16 Operator Precedence and Indentation

Having introduced all the operators in TC, it remains to show how operators are grouped into expressions. Expression grouping in TC is determined by three kinds of rules: (1) precedence and associativity, (2) indentation, and (3) explicit parentheses. This section describes these rules and how they interact to determine the parse of a TC expression.

For simplicity, all TC operators share the same level of precedence. Unlike arithmetic operators, whose precedence relationships were well-established long before being encoded in a programming language, the TC operators are generally unencumbered by convention. Also, unlike arithmetic, the "natural" precedence for an expression with natural language keywords seems to depend on a semantic interpretation of the expression. Consider the following expressions with identical operators but significantly different "natural" interpretations:

```
(1) Bag containing Apple or Orange containing Worm
(2) Bag containing Apple or Box containing Orange
```

The first expression might be naturally interpreted as "a bag with either an apple or orange in it that has a worm", while the second expression is more naturally interpreted as "either a bag containing an apple or a bag containing an orange". The first interpretation gives higher precedence to `or`; the second, to `containing`.

Rather than define an arbitrary precedence order which might seem natural in some cases but very unnatural in others, and which in any case would have to be memorized, TC takes the simple approach of no precedence at all.

Since TC has only one precedence level, the parse of an unparenthesized, single-line expression is determined by associativity. All operators in TC are right-associative, so that the expression

```
A before B containing C
```

is equivalent to the fully-parenthesized expression:

```
A before (B containing C)
```

Right associativity is actually a side-effect of the indentation rule, which is explained next.

In order to specify expression grouping without parentheses, a TC expression may be split onto multiple indented lines. Each indented line starts with an operator, and the indentation of this operator determines which subexpression is its left operand. In particular, the operator should be indented just beyond the first token of the expression that becomes its left operand. Continuing the previous example, if we want to write `(A before B) containing C` without using parentheses, the indented expression would be:

```
A before B
  containing C
```

Since `containing` is indented just beyond `A`, it captures all of `(A before B)` as its left operand, so the interpretation is `(A before B) containing C`.

If the second line were indented more deeply, as in:

```
A before B
           containing C
```

then `containing` would capture only `B` as its left subexpression, so the interpretation would be `A before (B containing C)`. Extending this rule to single-line expressions explains why all TC operators are right-associative. An operator on the same line as the previous token captures only the token to its immediate left, so that

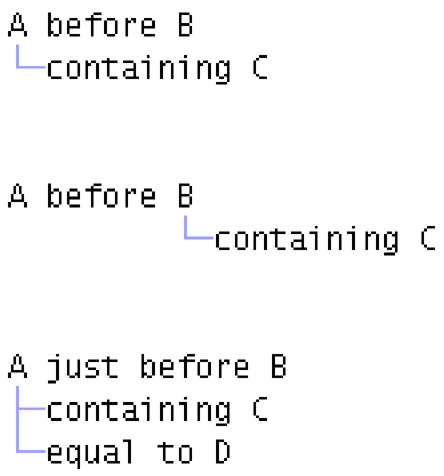
```
A before B containing C
```

is interpreted as `A before (B containing C)`.

A more precise definition of TC's indentation structure rules is deferred until the next section, which also presents an algorithm for parsing an indented expression.

Although indentation is used to signify block structure in other languages, like Python and Haskell, TC is unique in using indentation to structure binary infix expressions. The chief advantage of indentation in this context is that it allows a sequence of nested infix operators to be regarded as a sequence of constraints. For example, the indented expression

```
A just before B
  containing C
  equal to D
```



```
A before B
└containing C


A before B
         └containing C


A just before B
├containing C
└equal to D
```

Figure 6.8: The LAPIS pattern editor automatically displays light blue *tie lines* to indicate which token is modified by an indented expression.

can be regarded as a search for regions matching `A` that satisfy three additional constraints: `just before B`, `containing C`, and `equal to D`. The indentation structure shows the parallelism between the three constraints, and highlights the fact that the result of the pattern match will be regions from `A` (not `B`, `C`, or `D`).

LAPIS reinforces the visual meaning of indentation by drawing *tie lines* between an indented operator and the first token of its left operand. Figure 6.8 shows how LAPIS displays the examples shown above. As the user adjusts the indentation of a pattern in the LAPIS pattern editor, the tie lines are automatically redrawn to reflect how the pattern would be interpreted. Tie lines are a form of secondary notation that gives the user more feedback about the meaning of an expression.

Expression indentation has several tradeoffs, however. The first disadvantage is greater *viscosity*, the cost of making a change to an expression. Changing one line of an expression may require adjustments to the indentation of other lines too, in order to keep operators lined up with the appropriate tokens. Fortunately, this problem can be solved by environment support. The pattern editor in LAPIS automatically adjusts the indentation of subsequent lines as the user edits, in such a way that the expression structure is preserved. The pattern editor also has support for indenting and outdenting groups of lines simultaneously, in order to facilitate moving entire subexpressions up and down in the parse tree. These operations are described in more detail in the discussion of the LAPIS user interface (Chapter 7).

Attaching syntactic meaning to indentation also has the effect of making linebreaks syntactically significant, which in turn may lead to undesirably long lines. This problem is more serious in TC than in languages that use indentation only for block structure. In Python and Haskell, which use indentation only for block structure, a new block generally adds only 4 spaces (or at minimum, only 1 space) to the depth of indentation. In TC, however, the depth of indentation can depend on the length of the tokens and operators in the previous line. For example, suppose this expression is too long to fit comfortably on a line:

```
A contains B contains C contains D contains E contains F
```

There is no way to shorten this expression by inserting linebreaks, because the subsequent lines would have to be indented to nearly the same column positions as before:

```
A contains B
            contains C
                      contains D
                                contains E
                                          contains F
```

Other languages in which linebreaks are significant (such as Python and BASIC) deal with this problem with a special character that represents a continued line. In TC, rather than introduce a new special character, long lines can be split by adding explicit parentheses:

```
A contains B contains C contains (
    D contains E contains F
)
```

Parentheses (or curly braces, which are treated the same by TC) effectively block indentation grouping. An operator inside parentheses cannot capture an operand expression outside the parentheses.

Finally, a problem arises when tab characters are used in indentation, since tab widths can vary from editor to editor, changing the indentation and therefore the meaning of the expression. The pattern editor in LAPIS avoids this problem simply by avoiding tabs. Pressing the Tab key inserts only spaces, never tabs. Many other editors can be similarly configured. If an external editor is used to create a TC expression with embedded tabs, LAPIS uses a tab width of 8 to interpret the tabs. This tab width can be changed in the LAPIS preferences dialog. In general, however, the safest bet is to avoid tab characters entirely. A future version of TC may encourage this strategy by giving a warning when embedded tabs are encountered.

6.2.17 Parsing Indentation Structure

This section describes how TC expressions are parsed. The overall process has four phases:

1. Tokenization, in which the expression is divided into a sequence of tokens augmented by line and column position information;

2. Structuring, in which indentation and explicit parentheses are interpreted to connect the tokens into a tree;

3. Syntax checking, in which the token tree is checked for violations of syntactic and semantic rules; and

4. Translation, in which the token tree is translated into a syntax tree by a simple transformation.

Tokenization

Tokenization uses a conventional lexical analyzer to split the expression into tokens. Some operators in TC are represented by a multi-word phrase, such as `just before` or `overlaps start of`; these multi-word operators are represented by a single token after tokenization. However, multi-word operators in which the words are separated by other expressions use a different token for each component. For example, the `from-to` operator uses different tokens for `from` and `to`. All the synonyms for an operator are represented by a canonical token for that operator. Comments and whitespace are removed by the tokenization phase, but each token is marked with its line number and the starting and ending column position of the lexeme that generated it, so that indentation information can be recovered by the next phase.

Structuring

The structuring phase takes the sequence of tokens and connects it into a *token tree* representing the expression's structure. The token tree differs from a conventional syntax tree in two ways. First, the tree is arranged in *preorder*, so that the first token in an expression is at the root of the tree, rather than *infix order*, in which an operator would be at the root of the tree. Second, every token is a node of the tree, even if it is part of a multiple-token operator like `from-to`, `either-or`, or parentheses. In a conventional syntax tree, a multiple-token operator would occupy only a single node of the tree. An illustration of these differences can be found in Figure 6.9, which compares how arithmetic expressions parsed with conventional operator precedence would be represented as syntax trees and as token trees. Note that a token tree is not necessarily a binary tree, even if all the operators in the tree are binary.

The token tree representation is ideal for indentation parsing because it puts the first token of an expression at the root of its subtree. For any given token T, all tokens that appear to the right of T — either on the same line, or on subsequent lines indented deeper than T — become descendants of T in the token tree. Figure 6.10 shows the token tree for several TC expressions, in which the reader can readily verify this property. As a result, the token tree can be constructed by a simple algorithm that uses a stack to keep track of "open" tokens, i.e. tokens which appear above and to the left of the current line and column position. When a new token is returned by the tokenizer, any tokens that start to the right of the new token are popped off the stack. The top of the stack then becomes the new token's parent in the token tree.

Algorithm 6.1 illustrates this procedure. The algorithm has a wrinkle designed for smarter parsing of multi-token operators like `from-to`, `either-or`, and parentheses (lines 6–13). When the second token in a multi-token operator is encountered — for example, the `to` of a `from-to` expression — the algorithm looks down the stack (using the POPTO procedure) to find the corresponding first token, in this case `from`. If the matching `from` token is found, POPTO pops the stack so that `to` can be matched with `from`. This rule allows the algorithm to parse a one-line `from-to` expression, such as:

```
    from "Albuquerque" to "Phoenix"
```

Following strict indentation rules, the `to` token would be a child of `"Albuquerque"`, which would be a syntax error. Peeking down the stack to find the matching `from` token allows the system to interpret the expression more sensibly.

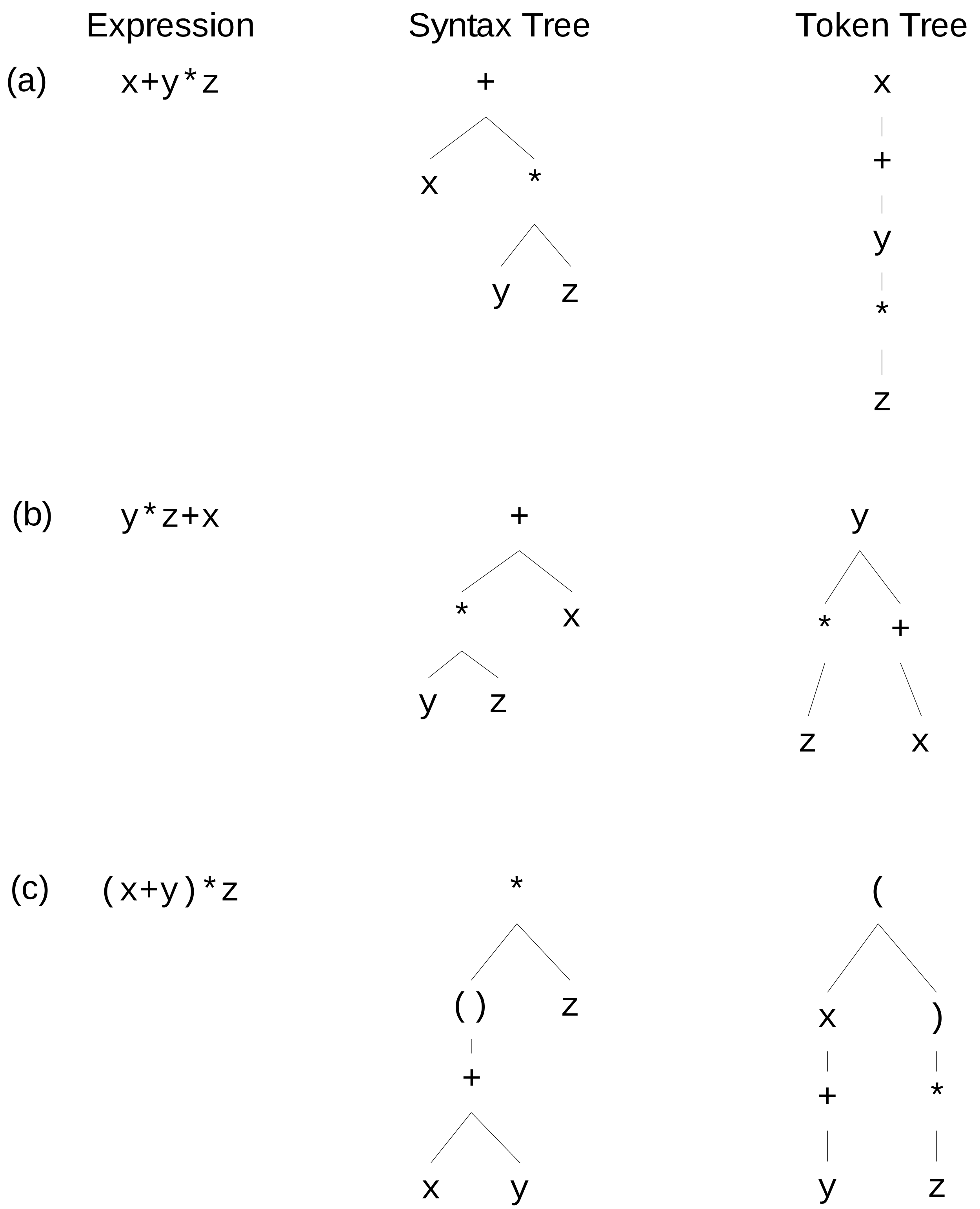


Figure 6.9: Comparison of token trees with syntax trees for several arithmetic expressions (parsed with conventional precedence, which gives multiplication higher precedence than addition).

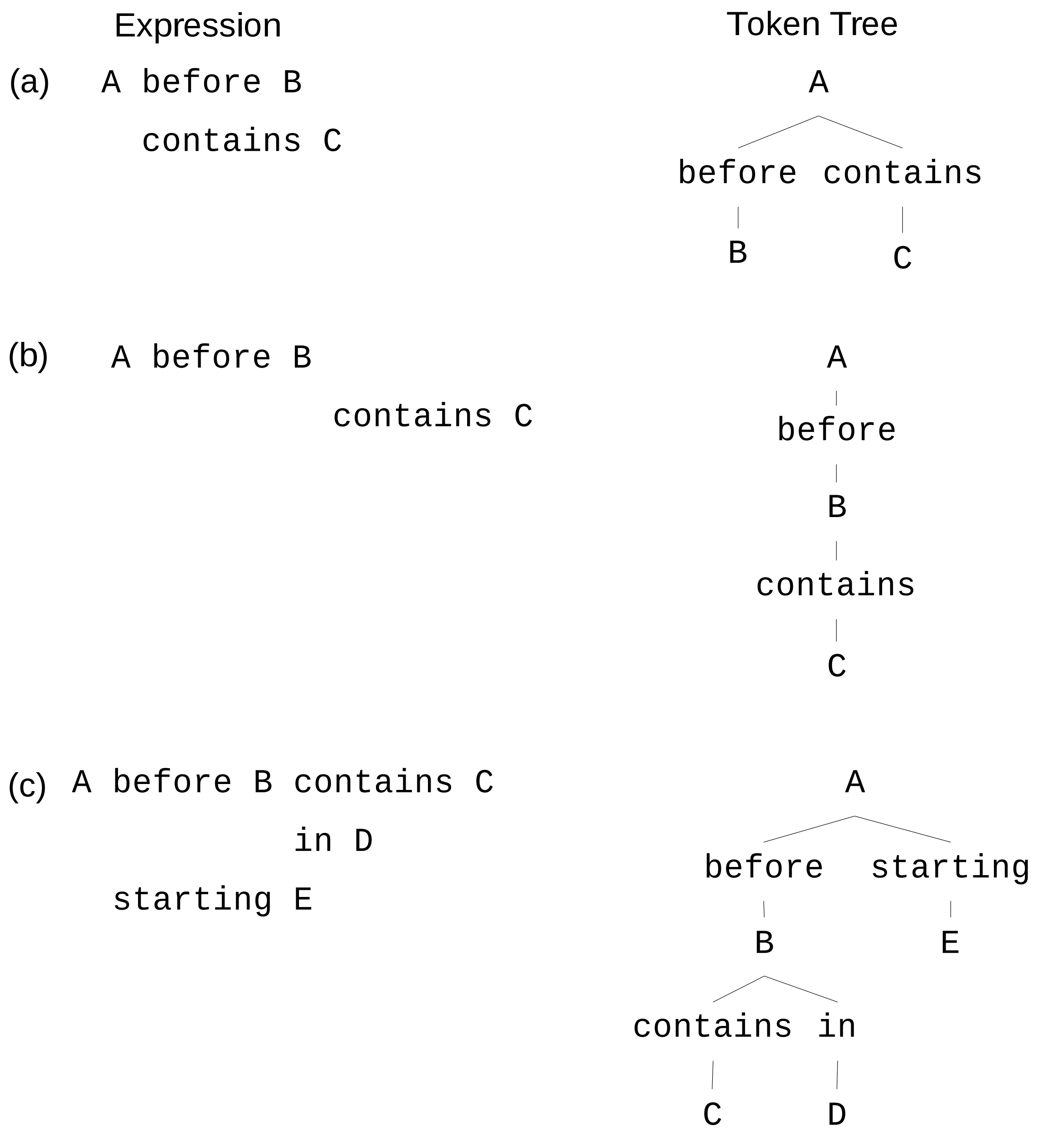


Figure 6.10: Token trees for several TC expressions.

If the stack does not contain a matching `from` token, however, PopTo does nothing, and the algorithm reverts to the parent indicated by indentation rules. Note that the structuring algorithm does not report any syntax errors. Syntax errors are delayed until the next phase of parsing. The tokenization and structuring algorithms are designed to always generate *some* token tree, even if it might be syntactically invalid, so that the token tree can be used to display tie lines (Figure 6.8) continuously while the user is editing a pattern.

Algorithm 6.1 ParseIndentation constructs the token tree for a TC expression from its indentation.

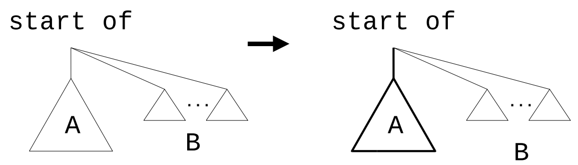
```
ParseIndentation()
 1  $stack \leftarrow$ new Stack
 2  Push($stack$, new Token( ROOT , $column = -1$))
 3  for each $t$ in token sequence from tokenizer
 4  do while Top($stack$).$column \geq t.column$ and Top($stack$) is not an unmatched ( or {
 5       do Pop($stack$)
 6       if $t =$ OR
 7          then PopTo($stack$, EITHER )
 8       else if $t =$ TO
 9          then PopTo($stack$, FROM )
10       else if $t =$ )
11          then PopTo($stack$, ( )
12       else if $t =$ }
13          then PopTo($stack$, { )
14       add $t$ as a new child of Top($stack$)
15      Push($stack, t$)
16

PopTo($stack, token$)
 1  for each $t$ in $stack$ from top of stack down
 2  do if $t = token$
 3        then pop off all tokens on stack above $t$
 4             return
 5  return (with stack unchanged)
```

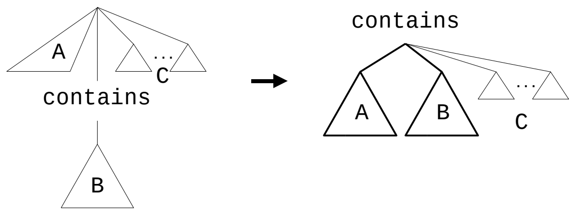
Parentheses and curly braces form a special case. The stack cannot be popped past a left parenthesis or curly brace until it has been matched by a right parenthesis or curly brace (line 4). When a right parenthesis or brace is encountered, it pops off all tokens until it reaches a left parenthesis or brace. Thus an expression inside parentheses or braces is completely isolated from the tokens outside it, so the indentation of the parenthesized expression can be independent of the expressions around it.

Syntax Checking

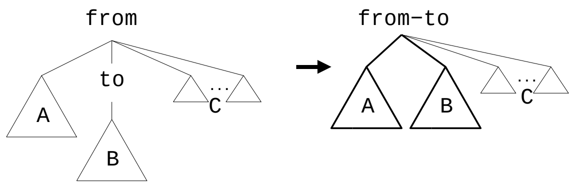
After the token tree has been constructed, it is checked for violations of syntax rules:



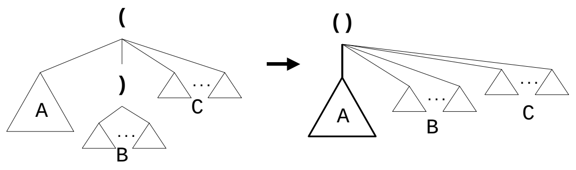
(a) unary operators



(b) binary operators



(c) binary operator with two tokens



(d) parentheses or curly braces

Figure 6.11: Rules for translating a token tree into a syntax tree. Light lines are token tree edges, and dark lines are syntax tree edges. Rules are applied to nodes in preorder, left-to-right, until all token tree edges have been transformed into syntax tree edges.

- operators with the wrong number of operands (e.g. `contains` dangling at the end of a line, with no right-hand side)
- incomplete multi-token operators (e.g. `from` without a corresponding `to`)
- unbalanced parentheses or curly braces
- operands of the wrong syntactic category (e.g. `prefix` and `is` require that one operand be an identifier)

All these tests can be made by a single pass over the token tree, checking local rules at every node. Failed tests display an error message to the user.

Translation

Finally, the token tree is transformed into a tree that can be evaluated, i.e., an abstract syntax tree in which the operands of each operator are its children. Although the token tree could conceivably be evaluated instead, such an evaluation would need to pass results *down* the tree as well as *up*. For example, evaluating the arithmetic expression token tree shown in Figure 6.9(b) would involve first computing the result of $y * z$ in bottom-up fashion, then passing that result down to the $+$ node to be added to x. Evaluating the syntax tree in Figure 6.9(b) is much simpler, because the evaluation is entirely bottom-up.

The transformation from a token tree to a syntax tree follows simple local rules. The rules for representative classes of operators are depicted in Figure 6.11.

```
Roundtrip Fares Departing From BOSTON, MA To
--------------------------------------------------
        $109            INDIANAPOLIS, IN
        $89             PITTSBURGH, PA

Roundtrip Fares Departing From PHILADELPHIA, PA To
--------------------------------------------------
        $79             BUFFALO, NY
        $89             CLEVELAND, OH
        $89             COLUMBUS, OH
        $89             DAYTON, OH
        $89             DETROIT, MI
        $79             PITTSBURGH, PA
        $79             RICHMOND/WMBG., VA
        $79             SYRACUSE, NY
```

Figure 6.12: Excerpt from an email message announcing cheap airfares.

6.3 Examples

This section gives some examples of using TC patterns to describe structure in text. The examples are divided into three kinds: plain text, web pages, and source code. More examples of TC patterns can be found in the LAPIS pattern library (Appendix A).

6.3.1 Plain Text

Patterns for plain text typically refer to delimiters like punctuation marks and linebreaks. Consider the following example of processing email messages. Several airlines distribute weekly email announcing low-price airfares. An excerpt from one message (from US Airways) is shown in Figure 6.12.

Describing the boundaries of the table itself is straightforward:

```
Table is from "Roundtrip Fares"
           to BlankLine
```

The rows of the table are just lines:

```
Flight is Line starting "$"
Fare is Number just after "$"
```

The origin and destination cities can be described in terms of their boundaries:

```
Origin is from AllCapsWord just after "From"
            to AllCapsWord just before "To"
            in Line starting Table
Destination is from AllCapsWord just after Fare
                 to end of Line
```

AlphaWindow,
Cumulus Technology Corp.,
1007 Elwell Court,
Palo Alto, CA, 94303,
(415) 960-1200,
$750,
Unix, **Discontinued**,
Alpha-numeric terminal windows, Window System

Altia Design, Altia,
5030 Corporate Plaza Dr \#300,
Colorado Springs, CO, 80919,
(800)653-9957 or (719)598-4299,
UNIX or Windows, IB

Amulet,
Brad Myers,
Human-Computer Interaction Institute,
Carnegie Mellon Univ,
Pittsburgh, PA, 15213,
(412) 268-5150,
amulet@cs.cmu.edu,
FREE,
X or MS Windows, portable toolkit, UIMS

Figure 6.13: Excerpt from a web page describing user interface toolkits.

Using these definitions, we can readily filter the message for flights of interest, e.g. from Boston to Pittsburgh:

```
Flight contains Destination contains "PITTSBURGH"
       in Table contains Origin contains "BOSTON"
```

The expression for the flight's origin is somewhat convoluted because flights (which are rows of the table) do not contain the origin as a field, but rather inherit it from the heading of the table.

6.3.2 Web Pages

Many web pages display data in a custom format, using HTML markup to set off important parts of the text typographically or spatially. Figure 6.13 shows part of a page describing user interface toolkits [Mye94].

The page describes over 100 toolkits with various properties: some are free, some are commercial; some run on Unix, others Microsoft Windows, others Macintosh, and others are cross-platform. To browse the page conveniently, the user might want to restrict the display to show only toolkits matching certain requirements – for example, toolkits running under both Unix and Microsoft Windows, sorted by price.

Each toolkit on this page is contained in a single HTML paragraph element. So the user might start by describing the toolkit as the Paragraph element, which is identified by the built-in HTML parser as `[P]`

```
Toolkit is [P]
```

Finding the prices is straightforward using Number, a pattern defined by the built-in US English parser:

```
Price is either "$" then Number
            or "FREE"
            in Toolkit
```

Finding toolkits that run under Macintosh is simple, since the page refers consistently to Macintosh as “Mac”. But Unix platforms are variously described as “X”, “X Windows”, or “Motif”, and Microsoft Windows is also called “MS Windows” or just plain “Windows”. One can deal with these problems by defining a pattern for each kind of platform that specifies all these possibilities and further constrains the matched literal to be a full word (not just part of a word):

```
Macintosh is Word = "Mac"
Unix is Word = either "Unix" or "X" or "Motif"
MSWindows is Word = either "PC"
                        or "Windows" not just after "X"
```

Using these definitions, the user can can readily filter the web page for toolkits matching certain requirements, e.g.:

```
Toolkit contains Unix
        contains MSWindows
```

for toolkits that run on both Unix and Microsoft Windows.

6.3.3 Source Code

Source code can be processed like plain text, but with a parser for the programming language, source code can be queried much more easily. LAPIS includes a Java parser, so the examples that follow are in Java.

Unlike other systems for querying and processing source code, TC operates on regions in the source text, not on an abstract syntax tree. At the text level, the user can achieve substantial mileage knowing only a few general types of regions identified by the parser, such as `Statement`, `Comment`, `Expression`, and `Method`, and using patterns to specialize them. For example, the Java parser in LAPIS identifies `Comment` regions, but does not specially distinguish the “documentation comments” that can be automatically extracted by Java’s `javadoc` utility. Figure 6.14 shows a Java method preceded by a documentation comment.

The user can find the documentation comments by constraining `Comment`:

```
DocComment is Comment starting "/**"
```

```
/**
 * Convert a local filename to a URL.
 * @param file File to convert
 * @return URL corresponding to file
 */
public static URL FileToURL (File file)
                    throws MalformedURLException {
    return new URL ("file:"
                    + toURLDelimiters
                        (file.getAbsolutePath ()));
}
```

Figure 6.14: A Java method with a documentation comment.

A similar technique can be used to distinguish public class methods from private methods:

```
PublicMethod is Method starting "public"
```

In this case, however, the accuracy of the pattern depends on programmer convention, since attributes like `public` may appear in any order in a method declaration, not necessarily first. All of the following method declarations are equivalent in Java:

```
public static synchronized void f ()
static public synchronized void f ()
synchronized static public void f ()
```

If necessary, the user can deal with this problem by adjusting the pattern (e.g., `Method starting Line contains "public"`) or relying on the Java parser to identify attribute regions (e.g., `Method contains Attribute equal to "public"`). In practice, however, it is often more convenient to use typographic conventions, like `public` always appearing first, than to modify the parser for every contingency. Since TC patterns can express such conventions, pattern matching might also be used to enforce them, if desired.

One can use `DocComment` and `PublicMethod` to find public methods that need documentation:

```
PublicMethod not just after DocComment
```

Java documentation comments can include various kinds of fields, such as `@param` to describe method parameters, `@return` to describe the return value, and `@exception` to describe exceptional return conditions. These fields can be described by TC expressions:

```
DocField is from "@"
              to end of Line
              in DocComment
ParamDoc is DocField starting "@param"
ReturnDoc is DocField starting "@return"
ExceptionDoc is DocField starting "@exception"
```

Using this structure, one can find methods whose documentation is incomplete in various ways. For example, this expression finds methods with parameters but no parameter documentation:

```
PublicMethod contains FormalParameter
             just after DocComment
                          not containing ParamDoc
```

Chapter 7

LAPIS User Interface

The ideas embodied in this thesis are implemented in a user interface called LAPIS. LAPIS combines a web browser and a plain-text editor into a single user interface, so that all three kinds of text relevant to this thesis can be edited and manipulated: web pages, plain text, and source code.

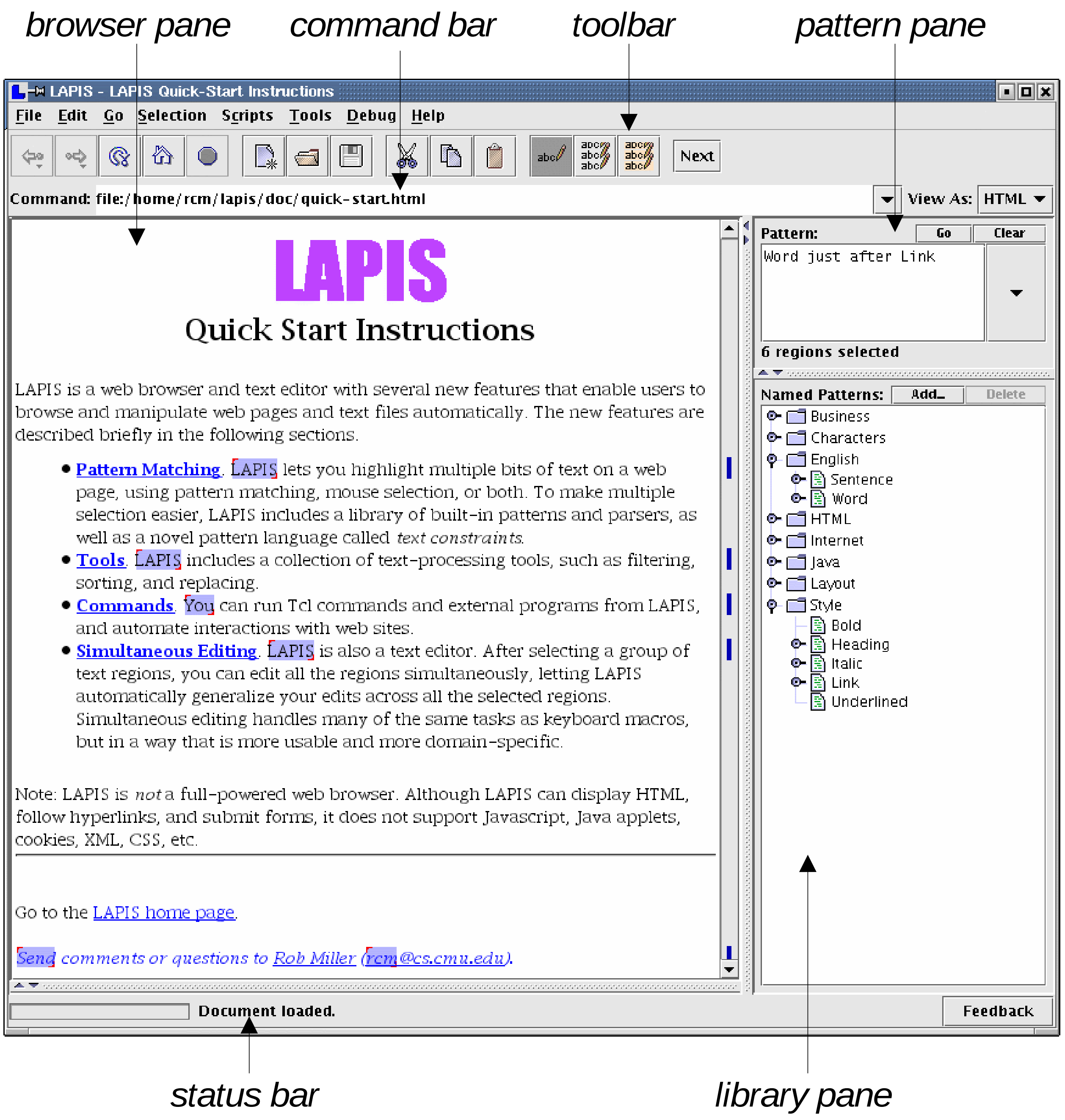
LAPIS has a number of interesting and novel features, the discussion of which will occupy the remainder of this dissertation. The current chapter describes multiple selections — specifically, how the user can make multiple selections with pattern matching or the mouse, how multiple selections are rendered on the screen, and how multiple selections are used for editing. The chapter concludes by describing a user study of multiple selections in LAPIS, focusing in particular on making selections with TC patterns.

Later chapters discuss the other important features of LAPIS: Unix-style tools and scripting (Chapter 8), inferring multiple selections from examples (Chapter 9), and finding outliers in multiple selections (Chapter 10).

7.1 Overview of LAPIS

A screenshot of LAPIS is shown in Figure 7.1. The LAPIS window has several main components:

- the *browser pane*, which displays a rendered HTML page or a text file.
- the *command bar*, where the user can enter the URL of a web page or file to load. As its name suggests, the command bar can also interpret typed commands in a scripting language (Tcl), which is described in more depth in Chapter 8. At the end of the command bar is the *View As* control, which allows the user to toggle the view back and forth between rendered HTML and plain text.
- the *toolbar*, which has not only the typical browser controls (Back, Forward, Reload, Home, and Stop) but also file-editing controls (New, Open, Save; Cut, Copy, Paste). The toolbar also has three buttons that control the selection inference mode (Manual, Guessing, and Simultaneous Editing), which are described in more depth in Chapter 9.
- the *pattern pane*, in which the user can type and execute TC patterns. The pattern pane is also used to display patterns automatically generated by other components of LAPIS, such as selection inference.



LAPIS

Quick Start Instructions

LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel pattern language called *text constraints*.
- **Tools**. LAPIS includes a collection of text-processing tools, such as filtering, sorting, and replacing.
- **Commands**. You can run Tcl commands and external programs from LAPIS, and automate interactions with web sites.
- **Simultaneous Editing**. LAPIS is also a text editor. After selecting a group of text regions, you can edit all the regions simultaneously, letting LAPIS automatically generalize your edits across all the selected regions. Simultaneous editing handles many of the same tasks as keyboard macros, but in a way that is more usable and more domain-specific.

Note: LAPIS is *not* a full-powered web browser. Although LAPIS can display HTML, follow hyperlinks, and submit forms, it does not support Javascript, Java applets, cookies, XML, CSS, etc.

Go to the LAPIS home page.

Send comments or questions to Rob Miller (rcm@cs.cmu.edu).

Figure 7.1: Screenshot of a LAPIS window showing its major components.

- the *library pane*, which displays the pattern identifiers that are defined in the global pattern library.

- the *status bar*, which displays progress messages and a percent-done indicator while LAPIS is downloading URLs, searching for patterns, or performing inference.

Although LAPIS can display both rendered HTML and plain text, its behavior is somewhat different in each case. When LAPIS is showing rendered HTML, it behaves like a web browser. The user can click on hyperlinks and fill out HTML forms, but cannot directly edit the content of the page. When LAPIS is showing plain text, on the other hand, it behaves like a text editor, allowing the user to insert and delete text. The user can toggle a page back and forth between the rendered, read-only view and the plain-text, editable view using the *View As* drop-down menu. A future version of LAPIS may lift this restriction, so that editing can be done in the rendered HTML view as well.

7.2 Multiple Selections

Unlike many web browsers and text editors, LAPIS permits arbitrary *multiple selections* to be made in the browser pane. Multiple selections are used for two purposes in LAPIS: to display the result of a pattern match, and to specify the arguments of editing commands (like Copy and Delete) and text-processing tools (like Extract and Sort). The screenshot in Figure 7.1 shows multiple selections: the word just after every hyperlink is selected.

Most browsers and editors support only one contiguous selection at a time. Some editors, notably Emacs and Microsoft Word, also allow *rectangular* selections, consisting of all the characters in a rectangle of row and column positions. The latest version of Microsoft Word, Word XP, supports multiple selections, but not multiple insertion points (zero-length selections).

Internally, the current selection is represented as an arbitrary region set, which may include nested or overlapping regions. This raises a number of questions:

- *Highlighting*: How should LAPIS display which regions are selected, if regions may overlap or nest?

- *Manual mouse selection*: How can the user add and remove selections from an arbitrary region set using the mouse?

- *Commands*: How should editing commands like Copy or Delete behave on an arbitrary region set?

The following sections give some partial answers to these questions. Generally, however, most of the design effort in LAPIS has been focused on the common case, in which the selection is a flat, non-overlapping region set. The user study described at the end of this chapter used only flat selections.

7.3 Highlighting Multiple Selections

This section describes how LAPIS highlights the current selection in the browser pane. For simplicity of presentation, the discussion is broken down by type of region set. First, techniques for highlighting flat region sets are presented. These techniques have been implemented in LAPIS and tested on users. Next is presented a technique for visually depicting the result of a unary relational operator, such as *containing* or *starting* or *just after*. This technique has been implemented in LAPIS but not tested on users. Then, some proposals for depicting nested and overlapping regions are presented, although these proposals have neither been implemented nor tested on users. The section ends by showing how the LAPIS scrollbar is augmented to indicate where selections are found in a long document.

7.3.1 Flat Region Sets

Even when the selection is a flat region set, LAPIS selection highlighting is subtly different from conventional text highlighting. Conventional text editors use a solid colored background that completely fills the selected text's bounds. Using this technique to highlight a flat region set has two problems.

First, two selected regions that are adjacent would be indistinguishable from a single selection that spans both regions. This problem is solved by shrinking the colored background by one pixel on all sides, leaving a two-pixel white gap between adjacent selections.

Second, two selections separated by a line break would be indistinguishable from a single selection that spans the line boundary. LAPIS solves this problem by adding small handles to each selection, one in the upper left corner and the other in the lower right corner, to indicate the start and end of the selection.

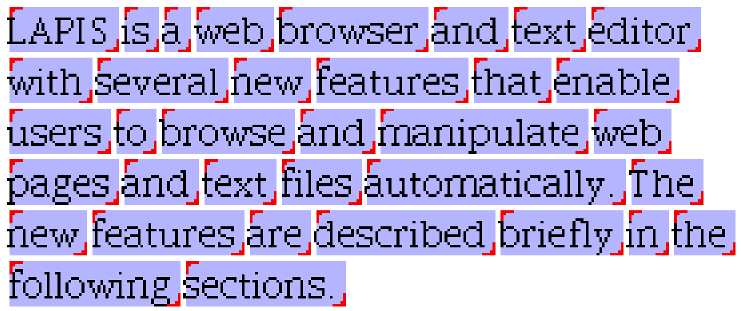
These small changes to conventional text highlighting allow any flat region set to be clearly and unambiguously rendered. Figure 7.2 shows how LAPIS highlights three different region sets in the same paragraph. Each of the region sets completely spans the given paragraph, in the sense that every character is included in some highlighted region. Thus conventional highlighting would highlight the entire paragraph in all three cases. In LAPIS, however, the white gaps and handles clearly mark the region boundaries, making it easy to distinguish the region set broken on word boundaries and the set broken on line boundaries from the set consisting of the entire paragraph.

7.3.2 Region Sets Created by Unary Relational Operators

A unique feature of the TC pattern language, relative to other structured text pattern languages, is that its relational operators are *unary*. The user can write a pattern like

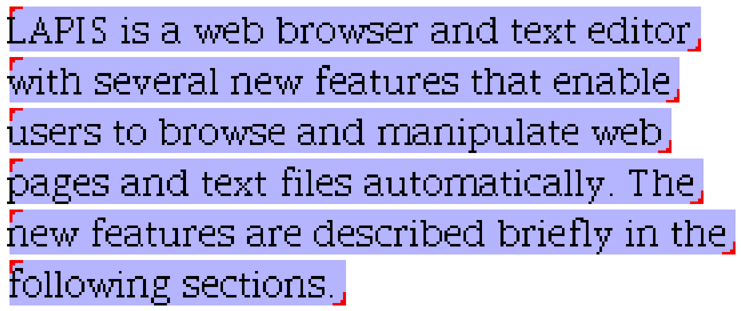
```
contains "web browser"
```

to match the set of all regions that contain an occurrence of the string "web browser". A pattern like this would probably not be useful for editing, since it is underconstrained. But the ability to visualize the region set matched by this pattern may still be useful, for two reasons. First, `contains "web browser"` may be a subexpression in a larger pattern that the user is trying to understand or debug. Second, unary relational operators are used as features for machine learning



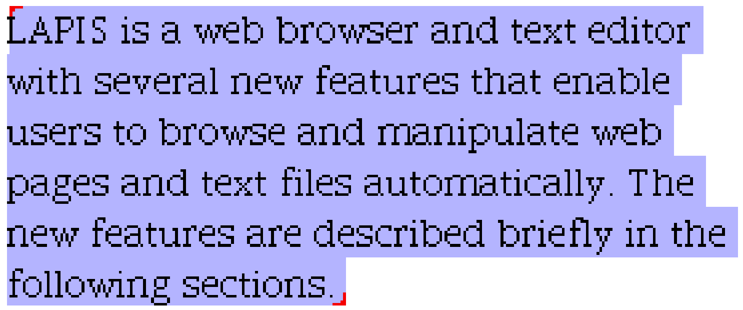
LAPIS is a web browser and text editor
with several new features that enable
users to browse and manipulate web
pages and text files automatically. The
new features are described briefly in the
following sections.

(a) words (including the space after each word)



LAPIS is a web browser and text editor
with several new features that enable
users to browse and manipulate web
pages and text files automatically. The
new features are described briefly in the
following sections.

(b) lines



LAPIS is a web browser and text editor
with several new features that enable
users to browse and manipulate web
pages and text files automatically. The
new features are described briefly in the
following sections.

(c) entire paragraph

Figure 7.2: Different region sets in the same paragraph, highlighted in LAPIS.

(Chapters 9 and 10), and the ability to display those features in the context of the document is another way to give the user feedback about how the system is learning.

With these goals in mind, a technique was developed that can visualize any region rectangle as a highlight in the browser pane. The highlight for flat regions described in the previous section is a special case of this technique. The technique is implemented in LAPIS, and it can display the results of unary relational operator patterns like the example shown above.

Recall from Section 4.1 that applying a unary relational operator like `contains` to a region produces a rectangle in region space. Each rectangle resulting from the relational operator is rendered by LAPIS using the following rules:

- characters included in every region in the rectangle are highlighted in dark blue;
- characters included in some but not all regions in the rectangle are highlighted in light blue;
- the boundary between dark blue and light blue highlighting is indicated by a smooth gradient;
- if all regions in the rectangle share the same start point (or end point), then that point is marked by a start handle (or end handle).

Figure 7.3 shows the effect of these rules on some common unary operators. For example, consider Figure 7.3(a). The characters in "web browser" are highlighted in dark blue, because every region that satisfies the condition `contains "web browser"` must include these characters. The characters outside "web browser" are also highlighted, but in light blue, because *some* region matching the condition includes them.

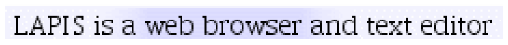
Figure 7.3(b), which shows the result of the `in` operator, is treated as a special case. Following the rules above, the highlight for `in` would simply be light blue, with no gradients. Since this would be indistinguishable from conventional text highlighting, however, LAPIS adds a gradient to each end of the highlight.

7.3.3 Nested and Overlapping Region Sets

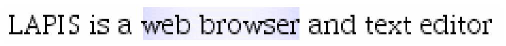
The highlighting technique described previously is used to highlight all region sets in LAPIS. Nested or overlapping regions are simply drawn on top of each other. The handles are always drawn last, so at least the endpoints of regions can be distinguished. Still, the results are not ideal. For example, Figure 7.4 shows that LAPIS highlights two different region sets, one nested and the other overlapping, in exactly the same way.

Solving this problem requires augmenting the highlight to distinguish nested regions from overlapping regions. Several alternative solutions are presented below and illustrated in Figure 7.5.

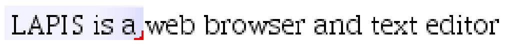
- **Underlining.** The region notation shown on the left side of Figure 7.4, which underlines each region separately, is clearly unambiguous. Inspired by that notation, highlights for nested and overlapping regions could be augmented with multiple underlines, as shown in Figure 7.5(a). Although this technique is unambiguous, it has two disadvantages. First, deeply nested regions may require more space between each line than is normally available, forcing the lines of text to move apart to accommodate a stack of underlines. Second, if the regions span multiple text lines, it would be hard for the user to follow the underlines across linebreaks.


LAPIS is a web browser and text editor

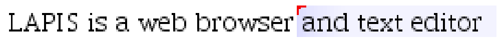
(a) contains "web browser"


LAPIS is a web browser and text editor

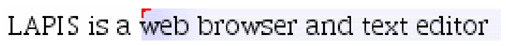
(b) in "web browser"


LAPIS is a web browser and text editor

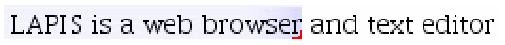
(c) just before "web browser"


LAPIS is a web browser and text editor

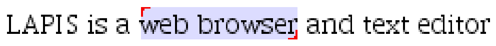
(d) just after "web browser"


LAPIS is a web browser and text editor

(e) starts "web browser"

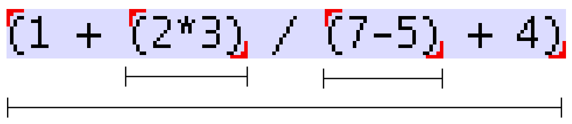

LAPIS is a web browser and text editor

(f) ends "web browser"

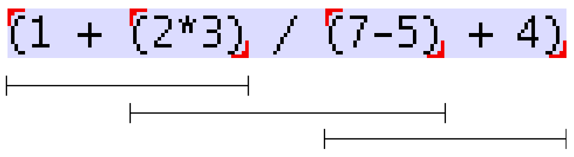

LAPIS is a web browser and text editor

(g) equals "web browser"

Figure 7.3: Gradient highlights displayed by LAPIS for common unary relational operators. (a) *contains* fades out in both directions away from the center; (b) *in* fades out inward to the center; (c) *just before* fades out to the left; (d) *just after* fades out to the right; (e) *starts* fades out to the right; (e) *ends* fades out to the left; and (f) *equals* has no gradient at all.

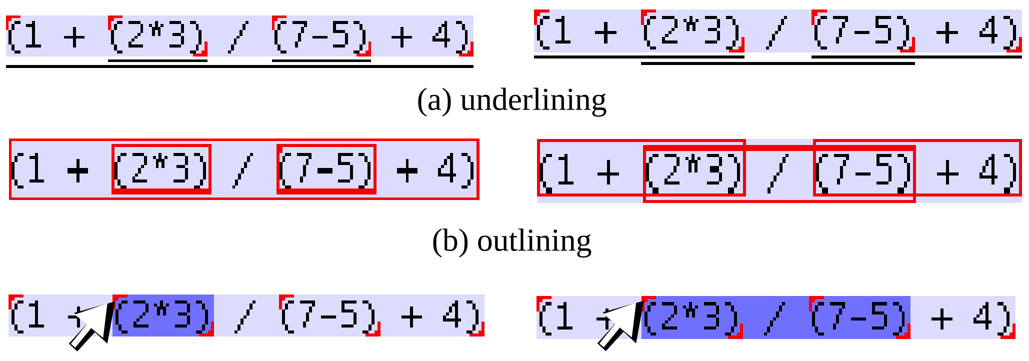

(1 + (2*3) / (7-5) + 4)

(a) nested


(1 + (2*3) / (7-5) + 4)

(b) overlapping

Figure 7.4: LAPIS highlighting cannot distinguish between nested and overlapping region sets.



(a) underlining

(b) outlining

(c) dynamic feedback

Figure 7.5: Techniques for distinguishing between nested and overlapping highlights.

- **Outlining.** Drawing an outline around each region is another way to distinguish between overlapping and nested regions. Figure 7.5(b) shows one way this outline might be rendered. Like underlines, however, outlines require extra space between lines of text to indicate deep nesting, and outlines may be hard to follow across multiple lines.

- **Dynamic feedback.** Unlike the previous techniques, this solution requires interaction from the user. When the user moves the mouse over a handle, the corresponding region receives some extra highlighting — perhaps deepening in saturation, acquiring a drop shadow or underline, or blinking briefly. Figure 7.5(c) shows what this might look like. This technique is probably far easier to implement than the previous ones, but it requires the user to actively explore the highlight in order to understand it. This technique also requires rendering multiple handles when several regions start or end at the same place.

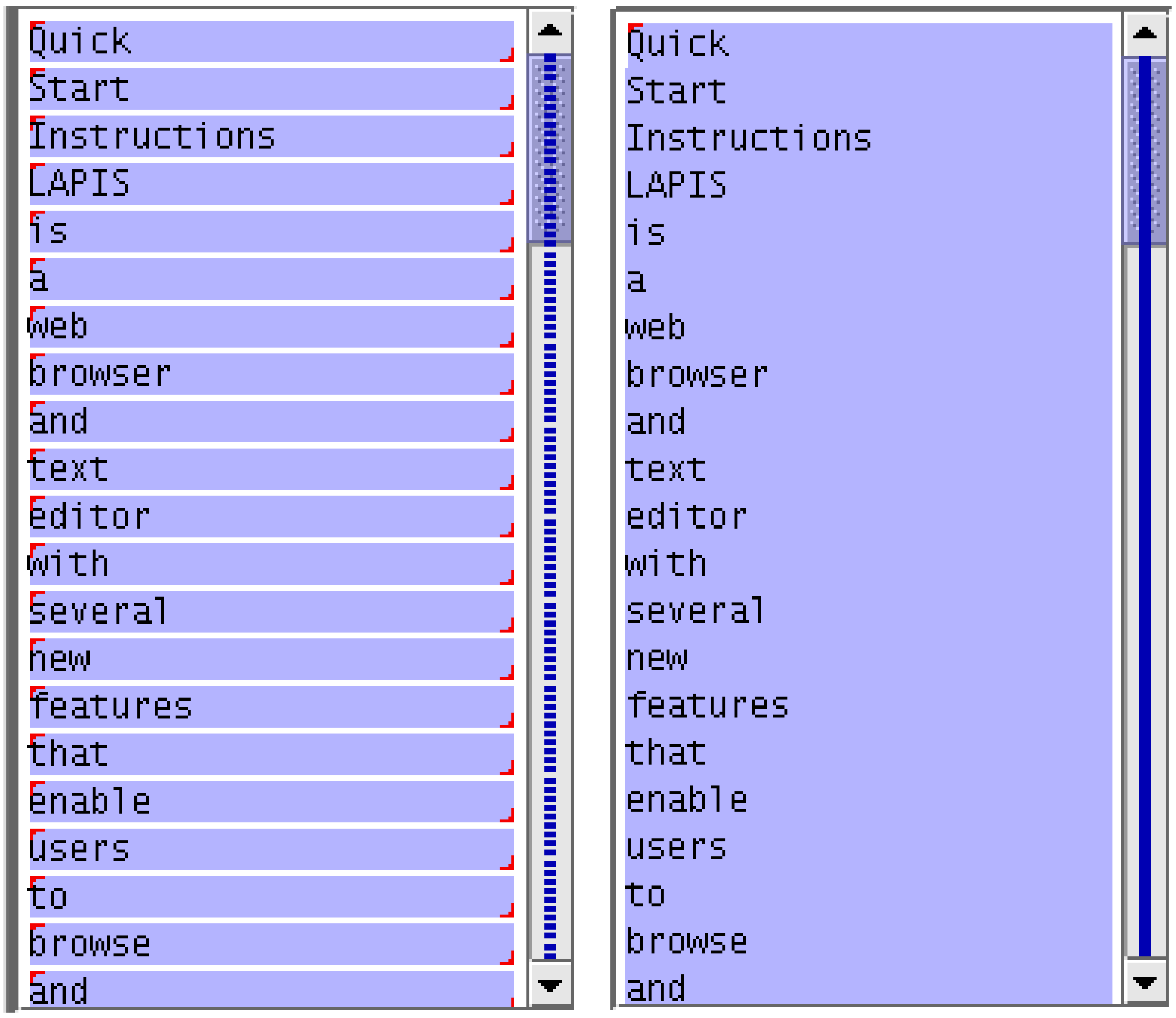
Other plausible ideas include color-coding or numbering of highlights and handles to distinguish them. Implementation and evaluation of these techniques are left as future work.

7.3.4 Scrollbar Augmentation

To help the user keep track of multiple selections in a long document, the browser pane's scrollbar is augmented with marks indicating where selections are located (Figure 7.6). Other systems have used the scrollbar for displaying secondary information about a document, such as bookmarks [Dan92] reading or editing frequency [HH92], and the current selection [MSC$^+$86].

Each mark drawn in the scrollbar corresponds to the vertical extent of a selected region. As in flat region highlighting, 1-pixel vertical gaps are left between marks from adjacent lines if sufficient resolution is available. This allows, for example, selections of individual lines (Figure 7.6(a)) to be distinguished from a selection that spans all the lines (Figure 7.6(b)). If the document is too long or the scrollbar too short, however, the vertical gaps are sacrificed, and regions from different lines may map to the same pixel in the scrollbar. Marks are drawn on top of the scrollbar's existing components, so that the scrollbar thumb does not obscure them, but as narrow rectangles, so that the thumb is not obscured either.

Scrollbar marks are also used in LAPIS to indicate the outliers in a selection (Chapter 10).



Quick
Start
Instructions
LAPIS
is
a
web
browser
and
text
editor
with
several
new
features
that
enable
users
to
browse
and

(a) lines

Quick
Start
Instructions
LAPIS
is
a
web
browser
and
text
editor
with
several
new
features
that
enable
users
to
browse
and

(b) entire document

Figure 7.6: The scrollbar is augmented with marks to help the user find selections in a long document. The marks can distinguish between different region sets, such as (a) and (b).

7.4 Making Multiple Selections

LAPIS provides four basic ways to set or modify the current selection: using the mouse; choosing a pattern identifier from the library pane; writing a pattern in the pattern pane, and inferring selections from examples. The first three techniques are described in the following sections. Discussion of the fourth technique, inference from examples, is deferred to Chapter 9.

7.4.1 Selecting Regions with the Mouse

Mouse selection in LAPIS generalizes the single-region selection techniques used in other text editors. When the left mouse button is pressed over the browser pane, the selection is first cleared. If the user drags the mouse with the left button held down, a single region is selected. An insertion point (zero-length region) is selected by clicking and releasing the left button. Keyboard keys can also be used to move the insertion point or make a single-region selection, following the usual conventions. For example, a single region can be selected by holding down Shift while moving the cursor with the keyboard arrows.

To add more regions to the selection, the user holds down the Control key while clicking and dragging. This use of Control as a modifier for discontinuous selection is also conventional, mimicking its use in file managers and drawing editors. Microsoft Word XP, which was released after LAPIS, also uses Control for making discontinuous selections. At one point in the development of LAPIS, the Shift key could also be used for the same effect, but several users objected because Shift is conventionally used to *extend* a single-region selection. As a result, the conventional behavior of Shift was restored, and only Control can be used to add more regions to a selection.

At present, mouse selection can only be used to add *flat* regions to the selection. If the new region overlaps an already-selected region, it is automatically merged with the existing selected region, rather than added as a new region. This decision was made for several reasons. First, flat selections are far more common in practice, particularly when the user is editing. Second, the current LAPIS highlighting is poor at displaying nested and overlapping region sets (Section 7.3.3), so it isn't clear that users would notice selection mistakes that accidentally introduced nesting or overlapping in a flat region set. Indeed, in the user study described below (Section 7.6), which used an early version of LAPIS in which mouse selections could add overlapping and nested regions, one user accidentally created overlapping regions and never noticed the mistake. LAPIS was subsequently redesigned to make the common case (flat region sets) the default. In the future, it might be useful to support an additional modifier combination, such as Control-Alt, which would allow users to make nested or overlapping selections.

Originally, the handles at the start and end of each selected region (e.g., in Figure 7.2) were interactive, so that the user could click and drag a handle with the mouse to resize the region. However, when this behavior was tested in the user study, it was found that users tended to click the handles by accident when trying to click next to the selected region. Since there are other ways to extend a selected region — for example, using Shift-click, or making an overlapping selection that is automatically merged with it — the handles were subsequently made noninteractive.

To remove a region from the selection, the user holds down Control and clicks on the selected region, essentially toggling it off. The user can also right-click on a region to bring up the context menu, which (in addition to other commands) includes an Unselect command that unselects the

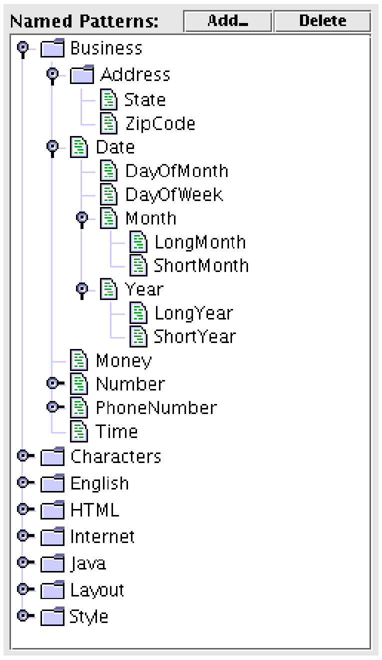


Figure 7.7: The library pane shows the patterns in the pattern library.

region that was clicked. The context menu also has an Unselect All command that clears all selections.

7.4.2 Selecting from the Library

The library pane provides another way to make a selection in LAPIS. The library pane displays the hierarchy of visible pattern identifiers in the pattern library using a tree widget. There are three kinds of items in the library pane:

- A *pure namespace* is a component of a namespace path. A pure namespace is denoted by a folder icon and a toggle switch that opens and closes its children, which are all the names in the namespace. Clicking on the name or icon also opens the namespace's children. All the top-level names, such as `Business`, `Characters`, and `English`, are pure namespaces.

- A *leaf* is a name bound to a pattern. It is denoted by a document icon filled with little marks that suggest highlighted pattern matches. Clicking on a leaf runs the associated pattern and selects the matching region set in the browser pane. In Figure 7.7, for example, `State` is a leaf name, which corresponds to the absolute pattern identifier `Business.Address.State`.

- A *bound namespace* is a namespace that is also bound to a pattern. A bound namespace is denoted by a document icon, like a leaf, but it also has children, like a namespace. In Figure 7.7, for example, `Date` is a bound namespace. Its absolute identifier, `Business.Date`, is bound to a pattern that matches entire dates, but it also acts as a namespace for other identifiers, such as `Business.Date.DayOfMonth,` which match parts of dates. Clicking on a bound namespace not only runs its associated pattern and selects the matches in the browser pane, but also opens its children.

Named patterns can be added and removed using the Add and Delete buttons. The Add button allows the user to name the current selection. Clicking Add brings up a dialog box in which the user can enter a name. If the current selection in the browser pane was produced by a pattern, then that pattern is assigned to the entered name, with the same effect as evaluating the TC expression `name is pattern` in the root namespace.

If the current selection was made by mouse selection, however, then LAPIS creates a *constant pattern* representing the selected region set. This constant pattern is not a TC pattern, but rather an internal Java object implementing the `Pattern` interface (Section 6.2.3) that encapsulates a region set and a pointer to the document. When the constant pattern is later applied to the same document, it returns the same region set. If the document has been edited since the constant pattern was created, the offsets specified in the region set are adjusted to account for the document changes using a coordinate map. (More about coordinate maps can be found in Section 6.2.12.) Since a constant pattern refers to specific character offsets, it is only useful on the document it was created for, so applying a constant pattern to a different document always returns the empty region set. To generalize mouse selections into a pattern that can apply to other documents, the user can use selection guessing (Chapter 9).

Selecting a name and clicking the Delete button removes the selected named pattern from the library. If the deleted name was a leaf, then it disappears from the library pane. If the deleted name was a bound namespace, then it becomes a pure namespace, having lost its pattern association. A pure namespace cannot be deleted in this way. A pure namespace does not disappear from the library pane until all of its children have been deleted.

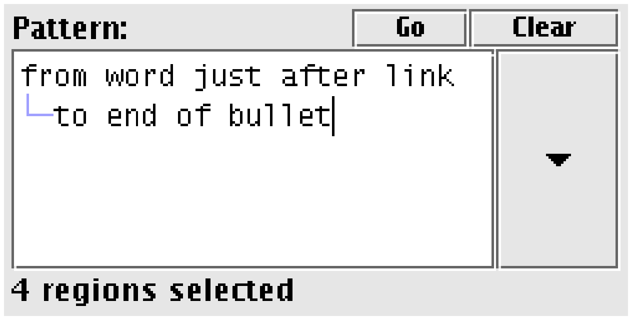
In the current version, library additions and deletions do not persist across different runs of LAPIS. Making library changes persistent currently requires changing the LAPIS configuration files by hand.

7.4.3 Selecting by Pattern Matching

The third way to make a selection in LAPIS is by writing a TC pattern in the pattern pane (Figure 7.8). Typing a pattern into this pane and pressing the Enter key or the Go button sets the current selection to all regions that match the pattern.

In order to accommodate multi-line TC patterns, the pattern pane is a multi-line text editor. Since Enter is used to run the pattern, linebreaks are inserted by pressing Control-Enter or Shift-Enter. Originally, these shortcuts were reversed: pressing Enter inserted a linebreak, and pressing Control-Enter (or the Go button) ran the pattern. In the user study, however, several users commented that using Control-Enter to run the pattern was counterintuitive — they expected to just push Enter to trigger the search, as in a Find dialog or a web search engine query form. Patterns entered interactively tend to be short anyway, so it makes sense to use the Enter key for the common case (evaluating a pattern) rather than the less common one (creating a multi-line pattern).

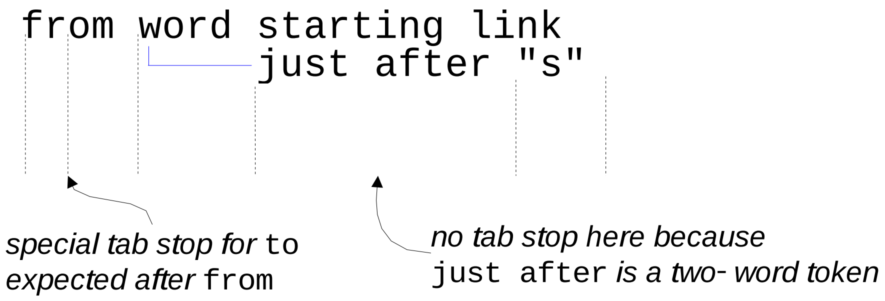
As shown in Figure 7.8, the pattern pane automatically displays *tie lines* to give feedback on how the system will parse the pattern. Tie lines are displayed by running the tokenization and structuring phases of TC expression parsing (Section 6.2.17) in the background as the user is editing the pattern. After each edit, the current expression is parsed into a token tree. The token



```
from word just after link
 to end of bullet
```

4 regions selected

Figure 7.8: The pattern pane allows the user to enter and run TC patterns.



```
from word starting link
          just after "s"
```

special tab stop for to expected after from
no tab stop here because just after is a two- word token

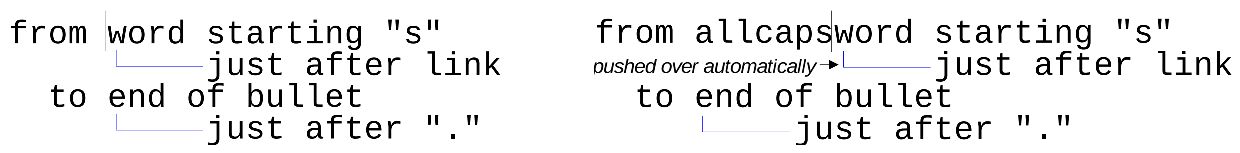
Figure 7.9: Adaptive tab stops.

tree is then used to draw a light blue line between the first token of each indented line (e.g., `to` in Figure 7.8) and its parent token in an earlier line (e.g., `from`).

Since indentation is essential to the interpretation of multi-line patterns, the pattern pane includes several features that make it easier to manage indentation. First, the Tab key does smart indentation, stepping through a set of adaptive tab stops that are determined by the possible parent tokens on earlier lines. Figure 7.9 shows an example. Most tokens produce one tab stop aligned with the start of the token, but tokens that are part of a multi-token expression, such as `from-to`, also produce a tab stop that would align the expected matching token, in this case `to`, with the end of the `from` token. To the right of the adaptive tab stops, the Tab key reverts to a fixed tab stop every 4 spaces. Shift-Tab can be used to move backwards through tab stops. If a group of lines are selected, then Tab and Shift-Tab adjust the indentation of the entire group, following the adaptive tab stops defined for the first line in the group.

In order to preserve expression structure when the user inserts or deletes pattern elements, the pattern pane automatically adjusts the indentation of lines below and to the right of the cursor. In particular, when an insertion or deletion changes the column position of a token, the pattern editor automatically reindents the token's children in the token tree in order to preserve their relative column positions. Figure 7.10 shows an example of automatic reindentation. Automatic reindentation makes TC indentation notation less *viscous* (Section 6.1) by allowing the user to make small changes to a pattern without having to fix the indentation manually.

The pattern pane provides a few other controls in addition to the pattern editor (Figure 7.8). The Go button runs the pattern, the same as pressing Enter. The Clear button clears the pattern editor. Finally, the large arrow on the right side of the pattern editor displays a drop-down history



```
from word starting "s"
          just after link
   to end of bullet
          just after "."
```

```
from allcapsword starting "s"
                  just after link
   to end of bullet
          just after "."
```

Figure 7.10: Automatic subexpression reindentation preserves expression structure during pattern editing.

menu showing recently-executed patterns, most recent first. Choosing a pattern from the history menu loads it into the pattern editor, where it can be edited if desired and then executed again.

The pattern pane is also synchronized with the library pane, in the following sense. When the user types a (valid) pattern identifier into the pattern pane and runs it, the identifier becomes selected in the library pane, to show the user where the identifier is located in the library. Namespace categories are opened and the library pane is scrolled as necessary to make the selected identifier visible. Conversely, when the user clicks on an identifier in the library pane, the identifier is loaded into the pattern pane, to show the user how to obtain the same selection using a pattern. If the identifier alone would be ambiguous, then the pattern pane displays the shortest unambiguous suffix of the identifier's canonical name. For example, if the library contains both `Programming.Verilog.State` and `Business.Address.State`, and the user clicks on the former identifier in the library pane, then the pattern pane would display `Verilog.State`, which is sufficiently long to disambiguate it from the other pattern named `State`.

7.5 Editing with Multiple Selections

Once multiple selections have been made, editing with them is a straightforward extension of single-selection editing. Typing a sequence of characters replaces every selection with the typed sequence. Pressing Backspace or Delete deletes all the selections if the multiple selection includes at least one nonzero-length region. If the selection is all insertion points, then Backspace or Delete deletes the character before or after each insertion point. Other editing commands, such as capitalization changes, are applied to each selection separately. If LAPIS were capable of editing rendered HTML as well as plain text, it would also include editing commands for changing paragraph styles and character styles that would be extended to multiple selections in the same way.

Clipboard operations are slightly more complicated. Cutting or copying a multiple selection puts a list of strings on the clipboard, one for each selection, in document order. If the clipboard is subsequently pasted back to a multiple selection with the same number of regions, then each string in the clipboard list replaces the corresponding target selection. If the target selection has more or fewer regions than the copied selection, then the paste operation is generally prevented, and a dialog box pops up to explain why. Exceptions to this rule occur when the source or the target is a single selection. When the source is a single selection, it can be pasted to any number of targets by replication. When the target is a single selection, the strings on the clipboard are pasted one after another, each terminated by a line break, and an insertion point is placed after each pasted string. Line breaks were chosen as a reasonable default delimiter. The user can delete the line breaks and replace them with a different delimiter using the new multiple selection.

For comparison, Microsoft Word XP supports a limited form of multiple-selection editing. Multiple selections can be created either with the mouse, by holding down Control while clicking and dragging, or by searching for a pattern, which can be either a literal string or a regular expression. Once multiple selections are created, the user can delete or apply a style command to all the selections. Multiple selections in Word XP do not act as insertion points for typing, however, nor can they be copied and pasted with the clipboard.

In LAPIS, if the selection contains nested or overlapping regions, those regions are flattened together before an editing command is applied to them. This design decision was taken for the same reason that mouse selection can only produce flat region sets — to avoid presenting a confusing editing model to the user. Editing with multiple nested selections may be useful for some tasks, particularly transformations on source code or XML, but the details and evaluation are left for future work.

Editing is not the only use that can be made of the selection in LAPIS. Chapter 8 describes some text-processing tools, such as Keep and Sort, that act on the current selection. Unlike editing commands, these tools are not limited to plain-text mode, but can also be used in the rendered HTML view.

7.6 User Study

After the initial design and implementation of the TC pattern language described in Chapter 6 and the LAPIS user interface described in this chapter, a small user study was conducted to evaluate their usability. The user study was designed as a formative evaluation, with the primary goal of discovering usability issues and resolving them to improve the language and user interface. Formative evaluations tend to answer qualitative questions like "Can a user do this kind of task using this feature?" and "What kinds of errors are commonly made?" Formative evaluations generally do not answer questions like "Is technique X more usable than technique Y?" More quantitative evaluations have been performed on some aspects of LAPIS, particularly selection inference (Chapter 9) and outlier finding (Chapter 10). Discussion of these other evaluations is postponed until those chapters.

7.6.1 Goals

The user study was primarily a test of the TC pattern language, since TC is the most complex piece of the user interface. Two questions are of chief interest:

- **Generation.** Given a region set in a document, can users write a TC pattern that matches it?
- **Comprehension.** Given a TC pattern, can users predict which region set it would match?

These questions guided the overall design of the experiment, which was split into two parts: generation and comprehension. In the generation part, the user wrote TC patterns to create certain selections in a document displayed by LAPIS. In the comprehension part, the user used a highlighting marker to make selections on paper. The tasks in each part were carefully chosen to test various features of the TC pattern language, and in particular to expose several hypothesized usability problems:

- **Literals.** Do users remember to quote literal strings?
- **Selecting attributes vs. whole objects.** One difference between TC queries and other kinds of queries is that a TC pattern specifies not only search constraints (e.g. `contains "Reynolds"`, or `not in Bold`) but also the space of text objects that should be searched (e.g. `Word`, or `Line`, or maybe `CourseTitle`). In other query systems, the query only specifies search constraints, and the object space is either fixed or specified some other way (e.g., by choosing a collection or database). For example, a web search engine searches for web pages, and a library catalog searches for books. Queries to these systems need not specify *what* the user wants to retrieve. TC patterns may return only certain attributes of the objects being searched — e.g., the titles of books matching the search constraints. Does this distinction cause confusion and erroneous patterns?
- **Operator names.** Do users use the relational operators correctly? Are any relational operators confused with each other? What other operator synonyms are suggested by common user mistakes?
- **Indentation.** Can users use indentation to structure complex expressions correctly?
- **Booleans.** Does the TC approach to Boolean operators (omitting `and` and encouraging an explicit `either` with each `or`) help users avoid common mistakes when formulating Boolean queries?
- **Abstractions.** Can users name patterns and use the named abstractions later? Do users name abstractions spontaneously, without being instructed to do so?

For reasons of time, the study did not comprehensively test *all* features of the TC language — only features that typical users would be likely to need during interactive use of LAPIS. In particular, advanced features like namespaces (Section 6.2.7), visible and hidden identifiers (Section 6.2.6), regular expressions (Section 6.2.10), and changing background (Section 6.2.14) were not tested by the study.

The study also did not ask users to generate or comprehend nested or overlapping selections, because those aspects of the LAPIS user interface are somewhat raw. Nevertheless, the study did attempt to test users' understanding of the underlying region set *model* that permits combinations of overlapping and nested regions:

- **Arbitrary region set model.** Can users comprehend patterns that combine two region sets that are not hierarchically compatible, e.g., lines and sentences?

In addition to addressing these questions about TC, the study also exercised other components of the LAPIS interface:

- **Highlighting.** Do users understand what region set is highlighted (particularly when some highlights are adjacent, as in Figure 7.2)?
- **Mouse selection.** Can users make multiple selections using the mouse? What typical errors are made? Do users combine mouse selection with pattern matching to make a difficult selection?

T1.	Highlighting
T2.	Patterns
T3.	Mouse Selection
T4.	Named Patterns
T5.	Naming a Pattern
T6.	Constraints
T7.	More Constraints
T8.	Either/Or
T9.	Not
T10.	Then
T11.	Counting
T12.	Summary

Figure 7.11: Tutorial sections.

- **Library pane.** Can users use the library pane to explore the pattern library and discover and use built-in patterns?

- **Pattern pane.** Can users use the pattern pane to enter, edit, and run TC patterns? What kinds of errors are made?

Multiple-selection editing was not tested in this user study, because it had not yet been implemented. Editing was tested in the later studies described in Chapters 9 and 10.

7.6.2 Procedure

The experiment consisted of four parts: pretest questionnaire, tutorial, generation, and comprehension.

The pretest questionnaire asked the user to evaluate his or her own computer experience in a number of areas: web browsing, word processing, search-and-replace, regular expressions, web search engines, and general computer programming. Users were also asked whether English was their native language, since TC uses English keywords and an English-like syntax.

The tutorial was a sequence of 12 web pages displayed in LAPIS, each of which described one feature of LAPIS or the TC pattern language. An outline of the tutorial is shown in Figure 7.11; the tutorial itself is included with the LAPIS distribution. Each page of the tutorial included one or more examples demonstrating the feature. Since the user viewed the tutorial inside LAPIS, the user was asked to try all such examples using the LAPIS user interface — for instance, typing example patterns into the pattern editor and running them to see what selection was made. The tutorial was designed to take about 30 minutes to complete. Afterwards, the user was provided with a printed copy of the tutorial, including a two-page summary of TC operators, which the user could consult as necessary during the remainder of the experiment.

The generation section consisted of 15 tasks. For each task, the user was asked to make a certain selection in a web page displayed in LAPIS. All tasks used the same web page, a short list of CMU computer science courses. Each task showed the desired selection marked in highlight marker on a printed copy of the web page, along with a brief English description of the selection.

An example of a generation task is shown in Figure 7.12. The English descriptions of all 15 tasks are shown in Figure 7.13.

As observed by Pane and Myers [PRM01], there is some danger that describing tasks in English will bias the test by inadvertently suggesting the correct answer. Nevertheless, some kind of intentional description of the desired selection seems unavoidable, particularly for complex Boolean queries. The English descriptions were carefully worded to avoid using TC operator names and library pattern names, instead using nouns and verbs appropriate to the semantic domain. For example, the description of task G5 uses the phrase "taught by Moore" instead of "*containing* Moore", which would be too suggestive of the correct TC operator. Similarly, G13 avoids use of the word "not". Some task descriptions must still use words that correspond to TC operators, but only when the meaning of the word in the description differs from its meaning in TC, so that a naive translation from English to TC would actually lead *away* from the correct pattern. For example, task G8 uses the phrase "in Wean Hall" to describe the location of the class, not of the desired selection. Naively translating this English phrase to the TC expression `in "WeH"` would not give the correct result; instead, the user must say *`contains`* `"WeH"`. Similarly, G12 uses "start at 10:30" in a sense different from the TC operator `starting`.

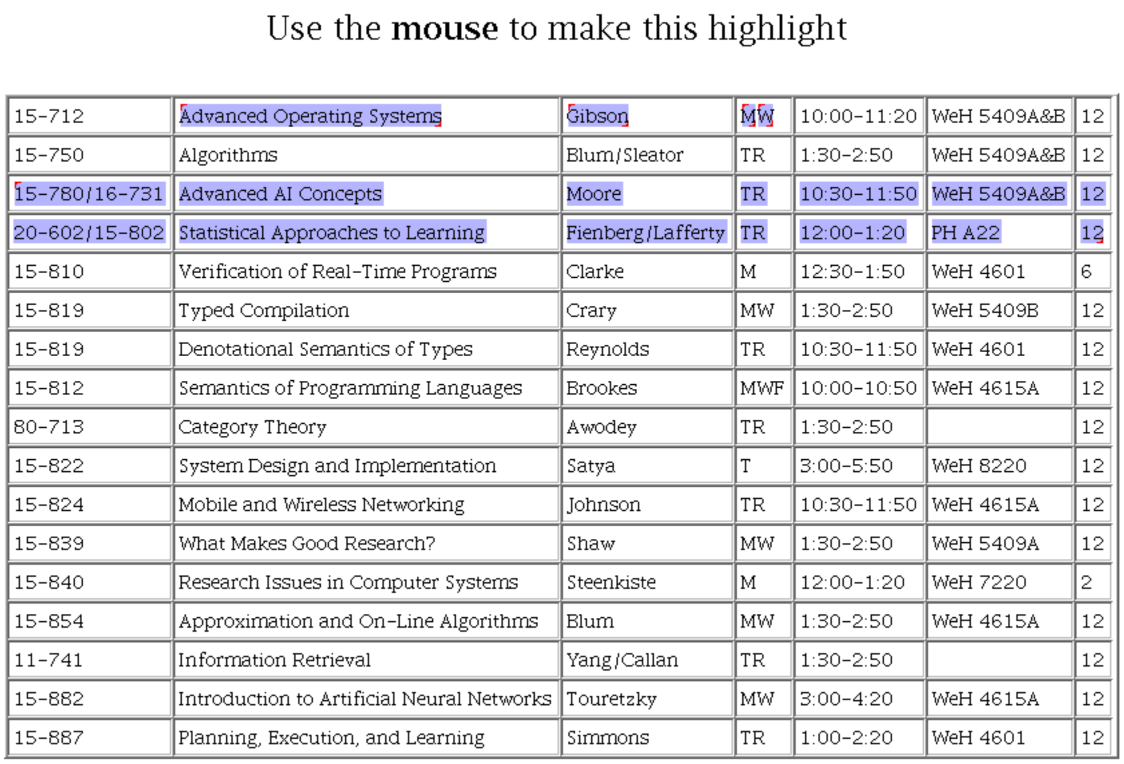
LAPIS provides several ways to make selections, and it was desirable to have users try all the methods in order to discover their usability problems. As a result, some of the tasks specify that the user should use a particular method to make a selection. For example, G1 must be done by writing a pattern, G2 using the mouse, and G3 by picking a pattern from the library. Some of the more challenging tasks (G10-G13) must be done by writing patterns, in order to force the user to try to write complex patterns. For some tasks, however, users were allowed to make the selection however they liked (G6, G7, G9, G14), in order to see which techniques users would prefer, and in particular whether users would combine pattern matching with mouse selection on the same task.

While the user was interacting with LAPIS, both in the tutorial section and in the generation section, the system recorded a transcript of selection-related user interface actions: TC patterns that were run, mouse selections that were made, and library names that were clicked on. Users were urged to "think aloud" as much as possible. The experimenter took notes about the user's verbal comments and the context in which they were uttered, but no audio or video recordings were made.

After the generation section came the comprehension section, which consisted of 10 tasks presented and performed entirely on paper. Each task showed a TC pattern, indented and augmented with tie lines as it would be displayed in the LAPIS pattern editor, and a printed copy of a web page. The user was asked to use a green highlighting marker to indicate what selection would be made in the web page if the given pattern were entered into LAPIS.

The 10 patterns used in the comprehension section are shown in Figure 7.14. Most of the tasks used the same web page as in the generation section, the listing of CMU computer science courses (Figure 7.15). Tasks C7 and C8 used a different web page, consisting of the first 8 sentences of the Gettysburg Address ((Figure 7.16). A different page was used for these two tasks so that the patterns could refer to two region sets which are not hierarchically compatible, specifically lines and sentences. Users were given a key that defined the named patterns used in the tasks (`Units`, `Row`, `StartTime`, `Line`, and `Sentence`) by showing the region set selected by each pattern.

The comprehension tasks were designed so that different ways of interpreting the pattern would result in different answers. For example, there are several plausible interpretations of C4, depending on how the constraint `starting "1"` is applied. The correct interpretation applies this


Use the **mouse** to make this highlight

15-712	Advanced Operating Systems	Gibson	MW	10:00-11:20	WeH 5409A&B	12
15-750	Algorithms	Blum/Sleator	TR	1:30-2:50	WeH 5409A&B	12
15-780/16-731	Advanced AI Concepts	Moore	TR	10:30-11:50	WeH 5409A&B	12
20-602/15-802	Statistical Approaches to Learning	Fienberg/Lafferty	TR	12:00-1:20	PH A22	12
15-810	Verification of Real-Time Programs	Clarke	M	12:30-1:50	WeH 4601	6
15-819	Typed Compilation	Crary	MW	1:30-2:50	WeH 5409B	12
15-819	Denotational Semantics of Types	Reynolds	TR	10:30-11:50	WeH 4601	12
15-812	Semantics of Programming Languages	Brookes	MWF	10:00-10:50	WeH 4615A	12
80-713	Category Theory	Awodey	TR	1:30-2:50		12
15-822	System Design and Implementation	Satya	T	3:00-5:50	WeH 8220	12
15-824	Mobile and Wireless Networking	Johnson	TR	10:30-11:50	WeH 4615A	12
15-839	What Makes Good Research?	Shaw	MW	1:30-2:50	WeH 5409A	12
15-840	Research Issues in Computer Systems	Steenkiste	M	12:00-1:20	WeH 7220	2
15-854	Approximation and On-Line Algorithms	Blum	MW	1:30-2:50	WeH 4615A	12
11-741	Information Retrieval	Yang/Callan	TR	1:30-2:50		12
15-882	Introduction to Artificial Neural Networks	Touretzky	MW	3:00-4:20	WeH 4615A	12
15-887	Planning, Execution, and Learning	Simmons	TR	1:00-2:20	WeH 4601	12

Figure 7.12: Example of a generation task. All generation tasks used the same data (a table of CMU computer science courses), but the instructions at the top and the selected region set varied.

G1. Please use a **pattern** to highlight all the occurrences of **WeH**
G2. Use the **mouse** to make this highlight
G3. Use a **built-in name** to make this highlight
G4. (1) Use a **built-in name** to make this highlight; then,
(2) Name the highlight **Course**.
G5. Use only **patterns** to highlight **all the courses taught by Moore.**
G6. (1) Make this highlight however you like; then,
(2) Name the highlight **StartTime**
G7. (1) Highlight **the course titles** however you like; then,
(2) Name the highlight **Title**
G8. Use only **patterns** to highlight **all the titles of courses taught in Wean Hall (WeH)**
G9. Highlight **all the 12-unit courses** however you like
G10. Use only **patterns** to highlight **all the course numbers in Department 15**
G11. Use only **patterns** to highlight **all the 12-unit courses that meet in Wean Hall (WeH)**
G12. Use only **patterns** to highlight **all the courses that start at 10:30 or else meet in WeH 4601**
G13. Use a **pattern** to highlight **all the courses taught by anybody other than Clarke**
G14. Highlight **all the course numbers** however you like
G15. Use only **patterns** to highlight **all the courses meeting in WeH 5409 starting at 10:30 or 1:30**

Figure 7.13: Generation tasks.

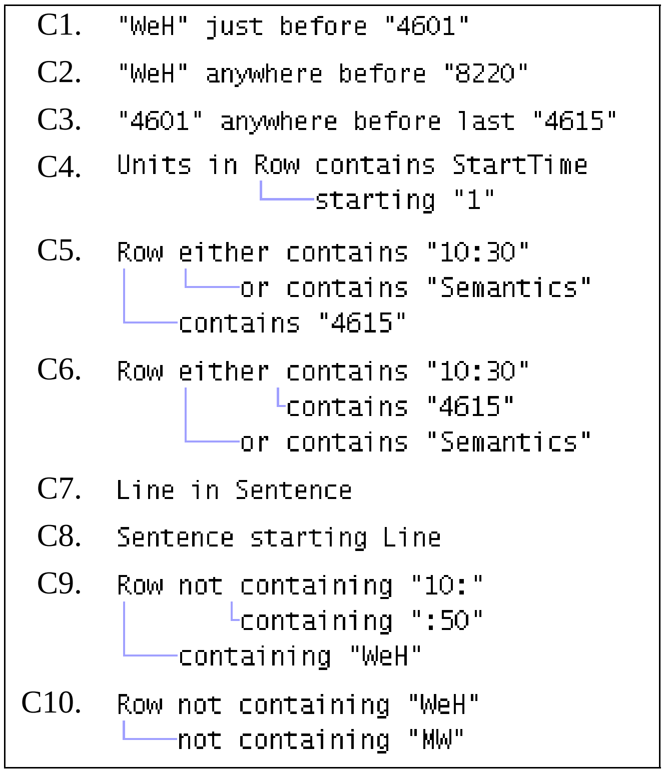
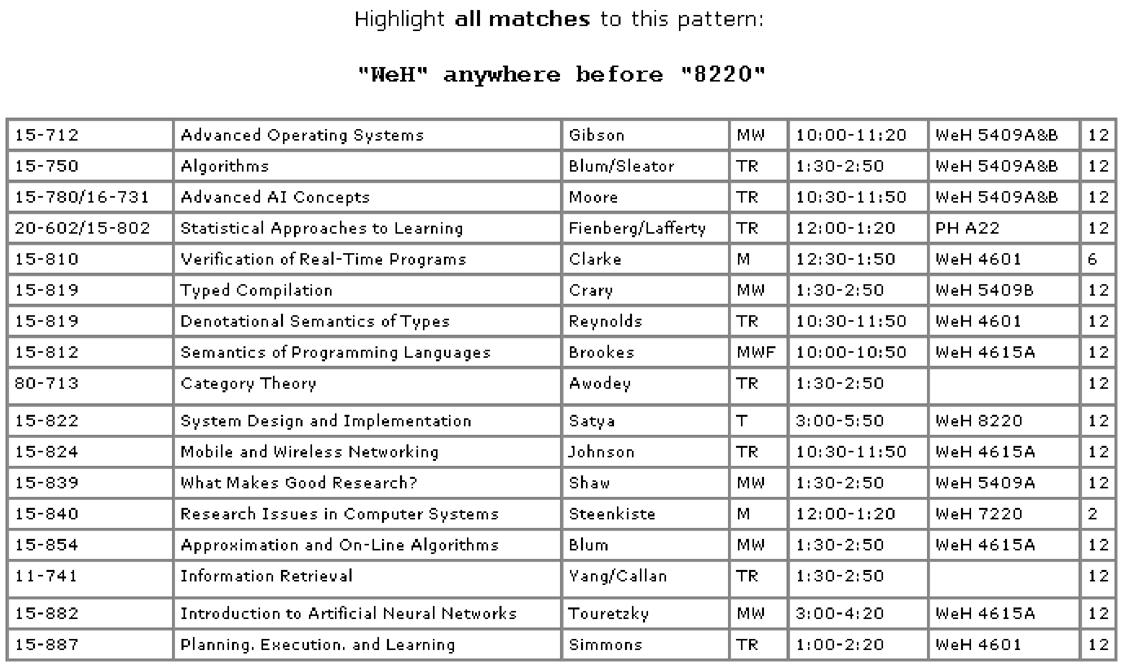

```
C1.  "WeH" just before "4601"
C2.  "WeH" anywhere before "8220"
C3.  "4601" anywhere before last "4615"
C4.  Units in Row contains StartTime
                 └──starting "1"
C5.  Row either contains "10:30"
     │     └──or contains "Semantics"
     └──contains "4615"
C6.  Row either contains "10:30"
           │    └contains "4615"
           └──or contains "Semantics"
C7.  Line in Sentence
C8.  Sentence starting Line
C9.  Row not containing "10:"
     │       └containing ":50"
     └──containing "WeH"
C10. Row not containing "WeH"
     └──not containing "MW"
```

Figure 7.14: Comprehension tasks.

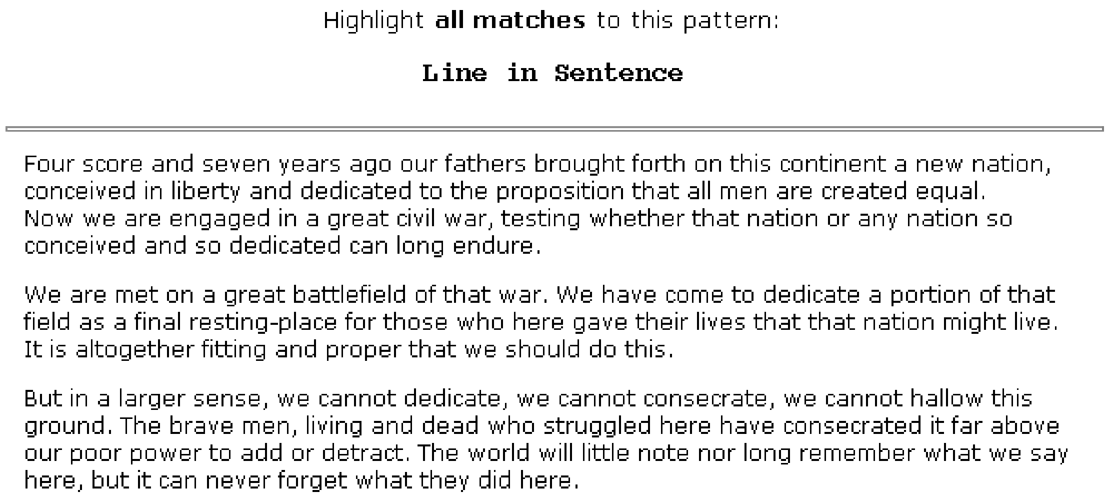


Highlight **all matches** to this pattern:

"WeH" anywhere before "8220"

15-712	Advanced Operating Systems	Gibson	MW	10:00-11:20	WeH 5409A&B	12
15-750	Algorithms	Blum/Sleator	TR	1:30-2:50	WeH 5409A&B	12
15-780/16-731	Advanced AI Concepts	Moore	TR	10:30-11:50	WeH 5409A&B	12
20-602/15-802	Statistical Approaches to Learning	Fienberg/Lafferty	TR	12:00-1:20	PH A22	12
15-810	Verification of Real-Time Programs	Clarke	M	12:30-1:50	WeH 4601	6
15-819	Typed Compilation	Crary	MW	1:30-2:50	WeH 5409B	12
15-819	Denotational Semantics of Types	Reynolds	TR	10:30-11:50	WeH 4601	12
15-812	Semantics of Programming Languages	Brookes	MWF	10:00-10:50	WeH 4615A	12
80-713	Category Theory	Awodey	TR	1:30-2:50		12
15-822	System Design and Implementation	Satya	T	3:00-5:50	WeH 8220	12
15-824	Mobile and Wireless Networking	Johnson	TR	10:30-11:50	WeH 4615A	12
15-839	What Makes Good Research?	Shaw	MW	1:30-2:50	WeH 5409A	12
15-840	Research Issues in Computer Systems	Steenkiste	M	12:00-1:20	WeH 7220	2
15-854	Approximation and On-Line Algorithms	Blum	MW	1:30-2:50	WeH 4615A	12
11-741	Information Retrieval	Yang/Callan	TR	1:30-2:50		12
15-882	Introduction to Artificial Neural Networks	Touretzky	MW	3:00-4:20	WeH 4615A	12
15-887	Planning, Execution, and Learning	Simmons	TR	1:00-2:20	WeH 4601	12

Figure 7.15: Example of the first kind of comprehension task. All but tasks C7 and C8 used this web page.



Highlight **all matches** to this pattern:

Line in Sentence

Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated can long endure.

We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting-place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

But in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow this ground. The brave men, living and dead who struggled here have consecrated it far above our poor power to add or detract. The world will little note nor long remember what we say here, but it can never forget what they did here.

Figure 7.16: Example of the second kind of comprehension task. Tasks C7 and C8 used this web page.

constraint to `Row`, in which case the user would highlight the units column in all rows that start with "1". There are two other plausible interpretations of this pattern, however: that the `Units` should start with "1", or that the `StartTime` should start with "1". Each of these interpretations produces a different highlight, so that particular misinterpretations can be identified by looking at the user's answer.

For complex patterns involving literal strings (e.g., C5, C6, C9, C10), there is some danger that the user will make errors not because of pattern misinterpretation, but merely because the user has overlooked some occurrence of a literal string. To distinguish between these kinds of errors, and to encourage the user to be methodical, the user was asked to circle all occurrences of literal strings in the document (using a pencil or ballpoint pen instead of the highlight marker). In retrospect, it would have been better if the printed document for each task simply made all occurrences of the relevant literal strings obvious, perhaps by printing them in contrasting colors. Then the test would focus more on pattern comprehension than on the user's ability to scan text for string matches. Fortunately, no users' answers were uninterpretable due to overlooked occurrences of a literal string.

The study was run with 3 unpaid pilot users followed by 6 paid users. The paid users were found by advertising on campus newsgroups (`cmu.misc.jobs` and `cmu.misc.market`) for subjects with at least some word-processing and web-browsing experience. Paid users received $12 for about 90 minutes of work. Observations of both pilot users and paid users are combined in the discussion in the next section. From their answers to the pretest questionnaire, all users had substantial experience in web browsing and word processing, and a range of experience in programming: 6 users with “lots” of programming experience, 2 users with “some” experience, and 1 user with “little” experience. Only 2 of the 9 users were non-native speakers of English.

Since the user study was intended as a *formative* evaluation, to guide further development of LAPIS, some usability problems were addressed by making changes to LAPIS during the study, in hopes that observation of the remaining users would show whether the changes had the desired effect. More changes to LAPIS were made after the conclusion of the study. Both kinds of changes are discussed in the next section.

7.6.3 Results

The discussion of the study results is organized around the goals identified earlier (Section 7.6.1).

Literals

Users neglect to quote literal strings in TC patterns. Here are some examples of patterns generated by users that omitted quotes from a literal string (the code in parentheses is the generation task the user was trying to solve):

```
WeH (G1)
Row starting 15-780 (G5)
Title in Row containing WeH (G8)
Title just before Gibson (G8)
7th Cell in Course not containing 12 (G9)
```

This problem has been noted by authors of other programming languages [Bru97]. The solution used for TC is automatic single-word quoting, which was described in detail in Section 6.2.9. Automatic quoting was one of the changes introduced after the first 5 users. The tutorial was not changed, however, and subsequent users were not told that automatic quoting would be available. Nevertheless, automatic quoting successfully prevented errors for the remaining users, two of whom solved task G1 using the quoteless pattern `WeH`.

Object Space

Two tasks were designed to test whether users could generate and comprehend patterns in which the search constraints (predicates that control whether a region was selected) were distinct from the object space (the regions actually selected).

Generation task G8 asked the user to write a pattern selecting the titles (the object space) of all courses taught in Wean Hall (the search constraint). One correct pattern is

```
Title in Course containing "WeH"
```

where `Title` and `Course` were named by the user in earlier generation tasks. One might expect users who misunderstand this aspect of TC to write a pattern that puts `Course` or `"WeH"` as the first component. Seven out of nine users successfully generated a correct pattern on the first try. One of the remaining users started with

```
title just before "WeH"
```

which is incorrect because the course title is not adjacent to the building in each row, but nevertheless identifies the correct object space.

The last user started with

```
WeH in title
```

which seems to indicate an object space problem. This user continued experimenting with other patterns, including `Title before "WeH"`, `"WeH" in Course`, and `Title containing "WeH"`. The user's think-aloud comments suggested that he/she was looking for a way to "correlate fields within a course" and "look for WeH outside of this set [the set of titles]". The user finally achieved the desired selection by first naming the set of courses that meet in Wean Hall:

```
pippo is Course containing "WeH"
```

and then using this named pattern to select the titles of the set:

```
2nd Cell in pippo
```

The other task that tested the object space concept was comprehension task C4, which asked the user to highlight the units column (the object space) in rows that start with the digit "1" (the search constraint). Although not all users highlighted the *correct* set of units, all nine users highlighted only units — not rows or start times or the digit "1". This suggests that they correctly identified the object space for this pattern.

	Suggested synonyms	
Operator	Typed	Aloud
`in`	of, in each	belonging to, within
`contains`	containg	whose, including
`equals`	is	
`just before`		right before
`starts`	begin	starts with
`then`	and	extending to more stuff

Figure 7.17: Synonyms for TC operators suggested by users.

Operator Names

The most interesting discovery about operator names was users' tendency to misspell `containing`. Three of the first 7 users misspelled the keyword at some point, two of them writing `containg` and the third `containging`. After these errors, LAPIS was changed so that the official keyword described in the tutorial was `contains`, with `containing` accepted as a synonym, without any mention of this fact in the tutorial. Subsequent users did use the official keyword `contains` (27 out of 47 occurrences), but also `containing` (19/47) and `containg` again (1/47). After the study, LAPIS was further changed to permit `containg` as yet another synonym.

The generation tasks offered an opportunity to discover other synonyms that users might find natural, shown in Figure 7.17. Some of these synonyms were typed in as part of attempted patterns; others were uttered verbally in the user's think-aloud protocol. Except for `containg`, each of these synonyms was suggested by only one user. Several of these synonyms were already available in TC, such as `begin` for `starts`, and `right before` for `just before`. Others were added to TC as a result of the study, particularly `containg`. (The current list of synonyms can be found in Figure 6.6.) Other suggested synonyms were deemed too vague or ambiguous, particularly `of` and `whose`. It is interesting to note that `and` was proposed as a synonym for `then`, confirming previous studies that `and` is commonly used to describe sequencing in natural language descriptions of programming tasks [PRM01].

Several comprehension tasks were designed to test potential confusions between operators. In particular, tasks C1 and C2 test whether the user can distinguish between `just before` and `anywhere before`. All users correctly answered both tasks.

Indentation

Indentation is used to structure TC expressions, instead of explicit parentheses. The most telling test of indentation rules was comprehension task C4, which asked users to interpret the pattern:

```
Units in Row contains StartTime
         └───starting "1"
```

The key question in this pattern is where the constraint `starting "1"` should be applied. Three plausible interpretations are possible:

- the `Row` must start with 1 (the correct interpretation under indentation rules)

- the `StartTime` must start with 1
- the `Units` must start with 1

Each of these interpretations produces a different answer. In the study, 5 out of 9 users chose the correct `Row` interpretation, 4 out of 9 users chose the `StartTime` interpretation, and no users chose the `Units` interpretation. Evidently, indentation is not strong enough to overcome users' tendency to read from left to right.

In generation tasks, two users made errors by expecting TC operators to follow "natural" precedence and associativity rules. Some examples of these incorrect patterns follow:

```
2nd Cell in course not starting "Statistical"
7th Cell in Course not containing "12"
row ending "12" containing "WeH"
```

In all three patterns, the last operator on the line was meant to apply to the first token on the line, which must be done by indentation. The user who wrote the first two patterns never got the hang of using indentation for Boolean `and`, adopting the strategy of naming all subexpressions instead to avoid building complex patterns. The user who wrote the third pattern realized what was wrong in a few seconds and added the indentation needed to correct the pattern.

Because of indentation structuring, newlines are significant in TC patterns. Two users failed to understand this fact and generated erroneous patterns with embedded newlines but no indentation, such as:

```
row either containing "30"
or containing "WeH 4601"

Digits then "-" then Digits in
1st Cell in Row
```

Although indentation structuring has mixed results as far as eliminating precedence errors and comprehension errors are concerned, users were nevertheless largely able to use indentation to structure expressions. For the generation tasks G11, G12, and G15, each of which involved multiple relational operators and at least one Boolean operator (two in the case of G15), a majority of users structured their patterns using indentation: 7 of 9 users for G11, 6 of 9 users for G12, and 6 of 9 users for G15. The remaining users managed to write only single-line patterns for these tasks, which sometimes required naming subexpressions.

Booleans

Several comprehension tasks tested whether users could correctly interpret Boolean combinations of constraints. For tasks C5 and C6, which combined `and` with `or` in the same expression, the results were positive: 8 out of 9 users interpreted both these patterns correctly. The ninth user (the same user in both cases) apparently interchanged the meanings of `and` and `or`. For example, the logical formula represented by task C5 should be

```
(contains 10:30 or contains Semantics) and contains 4615
```

but this user interpreted it as

```
(contains 10:30 and contains Semantics) or contains 4615
```

The user did exactly the same interchange on task C6.

For task C9, which involved a Boolean `and` in the scope of a `not` operator, the results were more mixed. Only 4 out of 9 users interpreted this pattern correctly, as

```
not (contains 10: and contains :50) and contains WeH
```

The remaining 5 users distributed the `not` incorrectly:

```
(not contains 10:) and (not contains :50) and contains WeH
```

In the generation tasks, users were largely able to produce Boolean expressions using indentation, as described in the previous subsection. Several users complained about the absence of `and` from the language, however. As a result of these observations, `and` was added to TC, with a warning about its ambiguity (Section 6.2.15).

Abstractions

Generation tasks G4, G6, and G7 tested whether users could assign names to patterns (`Course`, `Title`, and `StartTime`) and then use the named abstractions in later patterns. All nine users were able to name patterns; 8 of 9 used the Add button to name all patterns interactively, while the ninth user used the `is` operator for all naming. Almost all users, 8 of 9, reused `Course` and `Title` in later patterns. The remaining user never referred to any of these named abstractions, instead treating each task as an independent problem to be solved.

Four users assigned names to patterns spontaneously, without being requested to do so by the task. Three of these users used naming to express a complex pattern as a sequence of simpler patterns. For example, one user solved task G11 by first assigning the name `12units` to the 12-unit courses, then writing the pattern `12units containing "WeH"` to constrain it to the ones that meet in Wean Hall.

Two users spontaneously assigned names to natural abstractions in the data, such as `Units`, `Teacher`, and `Coursenumber`, even when these abstractions were not strictly necessary to solve the task. It is interesting to note that only one of these users was a programmer; the other described his or her programming experience as “little”.

Region Set Model

In general, most users were able to conceptualize overlapping sentence and line regions and determine relationships among them. Comprehension tasks C7 and C8 tested this ability with TC patterns involving the nonhierarchically-compatible region sets `Line` and `Sentence`. Most users, 7 out of 9, correctly interpreted task C7, `Line in Sentence`. One user misinterpreted it to mean essentially `Line overlaps Sentence`, i.e., that every line which included some part of a sentence should be highlighted, and proceeded to highlight every line in the document. The remaining user’s answer is simply wrong; no simple hypothesis can explain the user’s mental model.

For task C8, 5 out of 9 users gave the correct interpretation. The other 4 users misinterpreted the pattern in the same way, as if `starting` implied not only coincident start points but also containment — an understandable model, since the only examples of `starting` in the tutorial happened to imply containment.

Highlighting

Generation task G2 (reproduced in Figure 7.12) specifically tested whether the LAPIS highlighting techniques were sufficient to delimit flat region sets. G2 includes examples of both problems mentioned in Section 7.3.1: two adjacent regions that must be separately highlight (M and W), and two lines that form a single region (the two fully-selected rows). All users understood and correctly reproduced the selection in G2, so flat region highlighting can be deemed a success.

On two occasions, however, users inadvertently created overlapping region sets. One user ran the pattern `Word then Word`, which produces an overlapping set, and was confused by the resulting highlight. Another user inadvertently made overlapping regions while selecting with the mouse, but failed to notice the resulting unusual highlight. More design is needed to properly visualize nested and overlapping region sets, as discussed above (Section 7.3.3).

Mouse Selection

Generation task G2 also tested whether users could use the mouse to select multiple selections, and indeed all users could.

In the version of LAPIS used in the study, the handles at each end of a selected region were interactive, so that the user could click and drag them to resize the region. Although one user took advantage of this feature to resize a region, the interactive handles caused errors for other users. Three users accidentally clicked and dragged on handles when they were trying to make a selection adjacent to a selected region. After the study, LAPIS was changed so that the handles are no longer interactive.

Only task G2 explicitly required users to use mouse selection, but users sometimes resorted to mouse selection to accomplish other generation tasks. The most troublesome tasks were G7 and G14. On task G7, three users used mouse selection to make the entire selection; the remaining users were able to come up with a pattern. On task G14, two users made the entire selection manually, three users made the entire selection with a pattern, but four users first wrote an almost-correct pattern, then made a few mouse selections to fix it up. Thus users can and do combine pattern matching with mouse selection to accomplish tasks like these.

Library Pane

Two generation tasks, G3 and G4, requested that the user find a built-in named pattern corresponding to the desired selection. All users solved G3 immediately, probably because the answer (`Business.Time`) was usually still visible in the library pane as a result of a tutorial example that used `Business.Address.State`. Task G4, however, required users to poke around in the library hierarchy, searching for some pattern that selects the rows of the table. Eight of the 9 users successfully found it under `Layout.Table.Row`. The remaining user clicked on `Table` several times, which selected the entire table, but failed to notice that `Table` could be expanded further and hence never saw the pattern `Row`. This user did not realize that some names in the library can be simultaneously *patterns* that produce a selection and *namespaces* that group together other patterns. In the terminology presented earlier, `Table` is a bound namespace. After the study, to help address this problem, the library pane was changed so that clicking on a bound namespace not only runs its pattern, but also expands its children, so that the user can see that it has both kinds of behavior at the same time.

Interestingly, one user did not bother exploring the library pane to find named patterns. Instead, this user simply typed a name that seemed sensible: `time` for G3, and `row` for G4. This strategy succeeded, and LAPIS not only made the correct selection, but also showed the user where the name was actually defined in the library's hierarchical namespace.

Pattern Editor

Users made no errors with the pattern editor, apart from the indentation and newline problems discussed earlier, which can be attributed more to the underlying TC language than to the editing environment. As described earlier, however, several users objected to the use of Control-Enter to run the pattern, so LAPIS was changed after the study so that pressing Enter runs the pattern instead.

7.6.4 Summary

Overall, the user study identified a number of usability issues in TC and LAPIS, some of which were addressed in subsequent redesign of features. The following changes were motivated by observations in the user study:

- **Forgetting to quote literals.** TC now has automatic single-word quoting.
- **Misspelling "containing".** The official keyword is now `contains`, and common misspellings are accepted as synonyms.
- **Inadvertently clicking on handles.** The handles of a selection are no longer interactive.
- **Inadvertently selecting overlapping regions.** Mouse selection now selects only flat regions.
- **Expert users want `and`.** TC now supports the `and` keyword, with a warning that can be toggled off.
- **Bound namespaces don't look expandable in the library pane.** Clicking on a bound namespace in the library pane now does two things: runs its pattern, and expands it to show its children.

The user study also confirmed that precedence, expression grouping, and Boolean logic are trouble spots in programming languages, and TC's novel features (indentation structure, eliminating `and`) seem to help, but are not a panacea for these problems.

Chapter 8

Commands and Scripting

LAPIS includes a number of text-processing commands that are enabled by lightweight structure, primarily because the pattern library and TC pattern language make it possible to describe command arguments simply and concisely. This chapter[1] describes these commands and gives some examples of how they are used.

Many of the commands described in this chapter are inspired by the suite of text-processing tools in Unix [KP84]. Unix tools, such as `grep` and `sort`, are designed to be as generic and task-agnostic as possible, so that tasks can be done by combining a few generic tools rather than writing a program from scratch. Unfortunately, the generic nature of existing Unix tools is also a weakness, because generic tools can make only limited assumptions about the format of the text. Most Unix tools assume that the input is divided into records separated by newlines (or some other fixed delimiter character). But this assumption breaks down for richly structured text, such as source code, HTML, and XML. To overcome this difficulty, the LAPIS versions of these tools allow the structure of the input to be described by region sets — using library patterns, TC patterns, mouse selection, inference from examples (Chapter 9), or some combination.

One interface to these tools is graphical, using menus and dialog boxes to choose a command, configure its options, and invoke it. However, an important lesson from Unix is that generic tools provide the greatest leverage when they can be glued together into scripts and pipelines, creating new tools. Taking this lesson to heart, LAPIS also embeds a scripting language, Tcl [Ous94], and makes its text-processing tools available as Tcl commands as well. Tcl was chosen partly because of its syntactic simplicity, and partly because a good Java implementation was available [DeJ98].

Tcl is also well-suited to interactive command execution. Like the Unix shell, LAPIS has an interactive interpreter that allows script commands to be entered on the fly. Unlike the Unix shell, however, the LAPIS interpreter does not use a typescript shell, but a *browser shell*. The browser shell is a novel interaction model that integrates the interpreter into a web browsing interface, using the LAPIS command bar to enter script commands, the browser pane to display output, and the browsing history to manage the history of outputs. Among the benefits of a browser shell is an interesting new way to assemble pipelines of commands interactively, described in more detail later in this chapter.

This chapter also describes some commands for interacting with web sites, so that a user can write Tcl scripts that browse the Web automatically — clicking on hyperlinks, submitting forms,

[1] Portions of this chapter are adapted from an earlier paper [MM00].

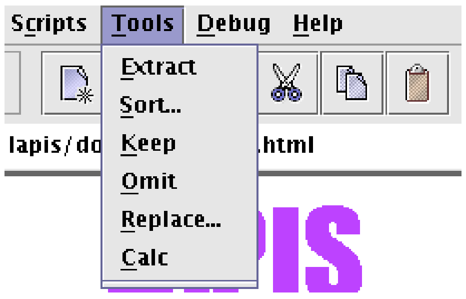


Figure 8.1: The Tools menu lists the text-processing tools built into LAPIS.

extracting data, and so forth. To create a browsing script quickly, the user can demonstrate it by recording a browsing sequence in LAPIS.

The chapter is sprinkled throughout with examples of scripts that use LAPIS commands to perform real tasks. An alphabetical reference to all LAPIS script commands can be found in Appendix C.

8.1 Text-Processing Commands

The Tools menu (Figure 8.1) presents a list of text-processing commands that can be applied to the current selection. Each menu command has an equivalent script command.

The menu commands are enabled only when at least one nonzero-length region has been selected in the browser pane. If the selection includes nested or overlapping regions, the selection is first flattened (Section 3.6.12), producing a flat region set, before being processed by the command.

All these commands produce a new document as output, rather than changing the current document. In that sense, they differ from commands in the Edit menu, like Cut, Copy, and Paste, which mutate the current document. For some tools, like Extract, generating a new document is appropriate; for others, like Sort, it may be better to change the current document, or at least give the user a choice. This design decision may be revisited in a future version of LAPIS.

After a command runs, its output document replaces the original input document in the browser pane. The original document is still available in the browsing history, so the user can use the Back button to undo the effects of the command.

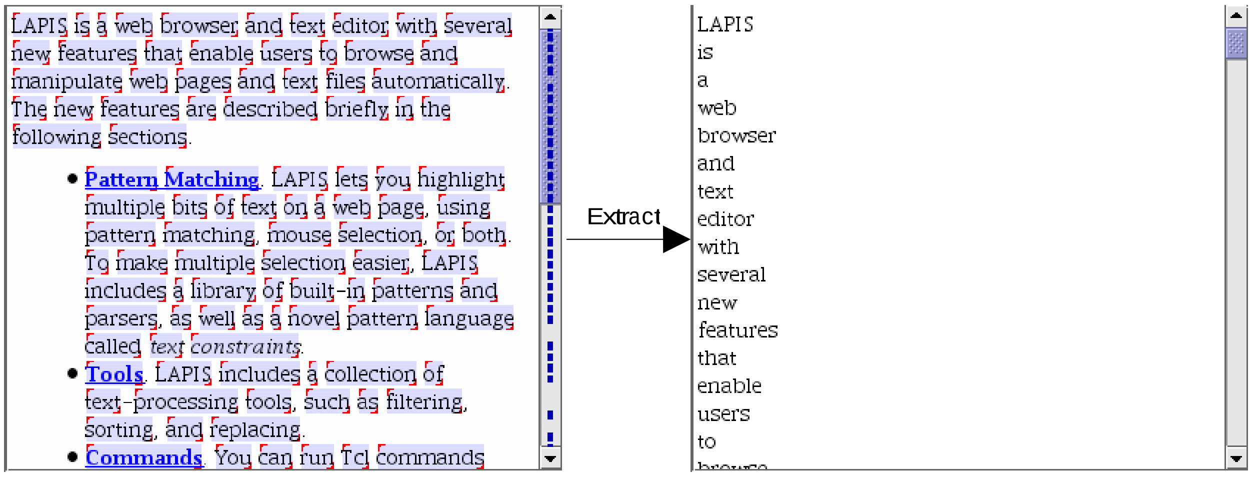
The next sections describe the current set of commands in LAPIS.

8.1.1 Extract

The Extract command extracts the regions in the current selection to produce a new document. The output consists of a list of items, one for each selected region. Figure 8.2 shows the results of applying Extract to various selections. The equivalent script command is

```
extract pattern
```

which extracts the region set matching a TC pattern. The named pattern `Selection` always returns the current selection in LAPIS, so a selection made with the mouse can be extracted by

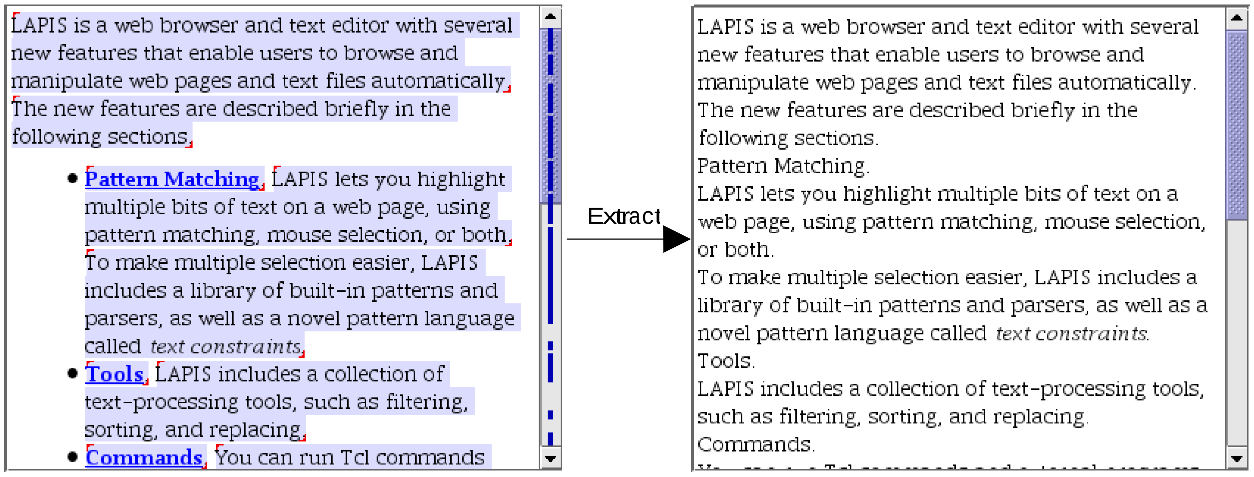


LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel pattern language called *text constraints*.
- **Tools**. LAPIS includes a collection of text-processing tools, such as filtering, sorting, and replacing.
- **Commands**. You can run Tcl commands

```
LAPIS
is
a
web
browser
and
text
editor
with
several
new
features
that
enable
users
to
```

(a) extracting words

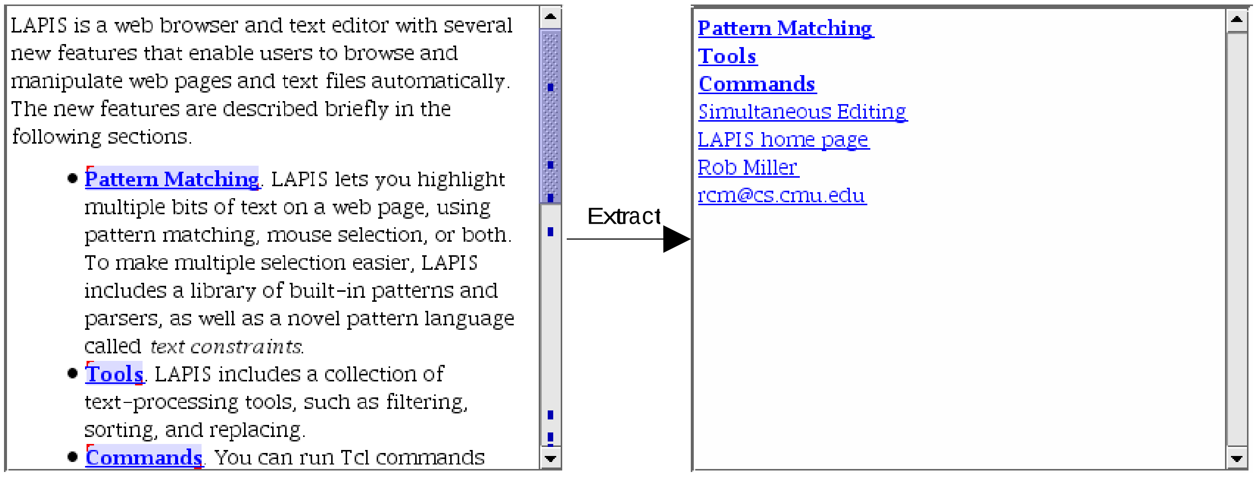


LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel pattern language called *text constraints*.
- **Tools**. LAPIS includes a collection of text-processing tools, such as filtering, sorting, and replacing.
- **Commands**. You can run Tcl commands

```
LAPIS is a web browser and text editor with several
new features that enable users to browse and
manipulate web pages and text files automatically.
The new features are described briefly in the
following sections.
Pattern Matching.
LAPIS lets you highlight multiple bits of text on a
web page, using pattern matching, mouse selection,
or both.
To make multiple selection easier, LAPIS includes a
library of built-in patterns and parsers, as well as a
novel pattern language called text constraints.
Tools.
LAPIS includes a collection of text-processing tools,
such as filtering, sorting, and replacing.
Commands.
```

(b) extracting sentences



LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel pattern language called *text constraints*.
- **Tools**. LAPIS includes a collection of text-processing tools, such as filtering, sorting, and replacing.
- **Commands**. You can run Tcl commands

Pattern Matching
Tools
Commands
Simultaneous Editing
LAPIS home page
Rob Miller
rcm@cs.cmu.edu

(c) extracting links

Figure 8.2: Examples of the output produced by Extract for various selections.

```
extract Selection
```

By default, Extract formats its output as a list of lines. The exact formatting depends on whether the input document is HTML or plain text. For HTML, Extract inserts a line break tag, `<br>`, after every extracted region. For plain text, Extract inserts a newline character after every extracted region, unless the extracted region already ends with a newline. This default formatting is simple, compact and works well enough in most cases. When extracted regions span multiple lines, however, it may be difficult to tell where one item begins and another ends (see, for example, the extracted sentences in Figure 8.2(b)).

For more sophisticated formatting, the user can specify the text that should be printed before, after, or between the extracted items. These delimiters can be specified by giving options to the `extract` command:[2]

```
extract pattern
        [-startswith start]
        [-endswith end]
        [-separatedby sep]
        [-as type]
```

Here, *`start`* specifies a string that should be printed before each extracted region, *`end`* is a string to be printed after each region, and *`sep`* is a string to be printed between each pair of regions. For example, the command

```
extract Word -startswith ( -endswith ) -separatedby ,
```

would print all the words in the input surrounded by parentheses and separated by commas, e.g.:

```
(LAPIS),(is),(a),(web),(browser),...,(cs),(cmu),(edu)
```

A solution to the problem of distinguishing multi-line extracted regions in HTML is to separate them with horizontal rule tags, `<hr>`:

```
extract Sentence -separatedby <hr>
```

Presently, delimiters can be specified only using the script command, in order to keep the Extract menu command simple.

The `extract` script command has one other option. The `-as` *`type`* option chooses the type of the output document, where *`type`* can be one of three choices: `text`, `html`, or `tcl` (a Tcl list). By default, the output type is the same as the type of the input document, which is either text or HTML depending on how the document is being viewed in the browser pane. Thus an HTML page whose source is being viewed as plain text is considered to have type `text`.

The output type determines two things:

[2]The square brackets in this syntax specification should not be typed as part of the command; they merely denote optional arguments to the `extract` command.

- Default output formatting in the absence of `-startswith`, `-endswith`, or `-separatedby` options. As explained above, the default formatting of HTML output prints a `<br>` tag after every region. The default formatting of text output prints a newline after every region that doesn't already end with a newline. The default formatting of Tcl output prints spaces between the regions, which is the format for a Tcl list.

- Quoting and type conversion. If the output type differs from the input type, then the Extract command automatically does whatever quoting is necessary to translate from one type to the other. Converting text to HTML replaces `<` with `&lt;`, `>` with `&gt;`, and `&` with `&amp;`. Converting from HTML to text deletes tags and expands entities like `&lt;` into the corresponding characters. Converting either text or HTML to Tcl produces a Tcl list — i.e., any output region which is not a simple word is surrounded by curly braces, `{}`, and any embedded occurrences of curly braces are escaped with backslashes.

The Tcl output type is useful for extracting all pattern matches as a list for processing by other Tcl commands. For example,

```
extract Sentence -as tcl
```

produces a Tcl list of all the sentences in the document.

The Extract command is not the only way to extract the current selection in LAPIS. The selection can also be cut or copied to the clipboard and then pasted back, into either the same document or a new document. Section 7.5 described in detail how this works. Note that the Copy command works in both the HTML view and the plain text view, but Paste only works in the plain text view at present, since the HTML view is not editable. Copying from HTML to plain text does the same conversions as Extract, deleting tags and expanding entities. With copy and paste, the user can control the destination of the extracted text by making multiple selections for the paste target. Copy and paste also allows the user to change the delimiters between extracted regions interactively. Figure 8.3 shows a sequence of editing commands including copy and paste that achieve the same effect as the `extract Word` command described above.

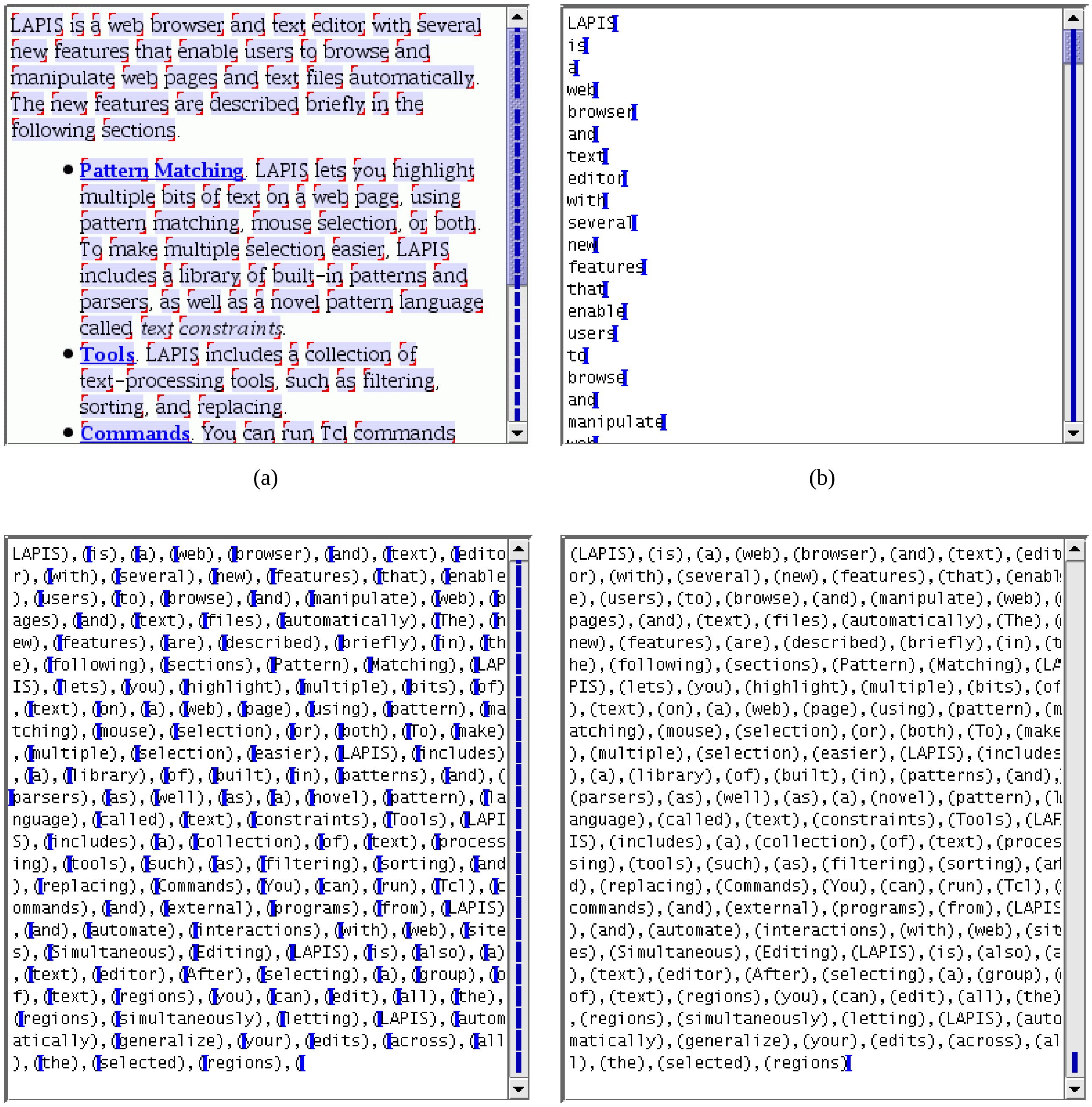
Extract is analogous to the Unix utility `grep`. Whereas `grep` can only extract lines, Extract can pull out an arbitrary set of regions matching a pattern, such as `Sentence containing "LAPIS"` or `MethodCall starting "printf"`. The Unix command

```
grep regexp
```

is roughly equivalent to the LAPIS command

```
extract {Line containing /regexp/}
```

(The curly braces are required by Tcl to quote a multi-word pattern as a single argument.) One must say "roughly equivalent" because the regular expression may be interpreted differently by `grep` and LAPIS. Regular expression languages differ, not only between LAPIS and `grep`, but also between different versions of `grep`. Also, regular expressions in LAPIS are not constrained to match within a line, as they are in `grep`.



(a)

LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel pattern language called *text constraints*.
- **Tools**. LAPIS includes a collection of text-processing tools, such as filtering, sorting, and replacing.
- **Commands**. You can run Tcl commands

(b)

```
LAPIS
is
a
web
browser
and
text
editor
with
several
new
features
that
enable
users
to
browse
and
manipulate
```

(c)

```
LAPIS),(is),(a),(web),(browser),(and),(text),(edito
r),(with),(several),(new),(features),(that),(enable
),(users),(to),(browse),(and),(manipulate),(web),(p
ages),(and),(text),(files),(automatically),(The),(n
ew),(features),(are),(described),(briefly),(in),(th
e),(following),(sections),(Pattern),(Matching),(LAP
IS),(lets),(you),(highlight),(multiple),(bits),(of)
,(text),(on),(a),(web),(page),(using),(pattern),(ma
tching),(mouse),(selection),(or),(both),(To),(make)
,(multiple),(selection),(easier),(LAPIS),(includes)
,(a),(library),(of),(built),(in),(patterns),(and),(
parsers),(as),(well),(as),(a),(novel),(pattern),(la
nguage),(called),(text),(constraints),(Tools),(LAPI
S),(includes),(a),(collection),(of),(text),(process
ing),(tools),(such),(as),(filtering),(sorting),(and
),(replacing),(Commands),(You),(can),(run),(Tcl),(c
ommands),(and),(external),(programs),(from),(LAPIS)
,(and),(automate),(interactions),(with),(web),(site
s),(Simultaneous),(Editing),(LAPIS),(is),(also),(a)
,(text),(editor),(After),(selecting),(a),(group),(o
f),(text),(regions),(you),(can),(edit),(all),(the),
(regions),(simultaneously),(letting),(LAPIS),(autom
atically),(generalize),(your),(edits),(across),(all
),(the),(selected),(regions),(
```

(d)

```
(LAPIS),(is),(a),(web),(browser),(and),(text),(edit
or),(with),(several),(new),(features),(that),(enabl
e),(users),(to),(browse),(and),(manipulate),(web),(
pages),(and),(text),(files),(automatically),(The),(
new),(features),(are),(described),(briefly),(in),(t
he),(following),(sections),(Pattern),(Matching),(LA
PIS),(lets),(you),(highlight),(multiple),(bits),(of
),(text),(on),(a),(web),(page),(using),(pattern),(m
atching),(mouse),(selection),(or),(both),(To),(make
),(multiple),(selection),(easier),(LAPIS),(includes
),(a),(library),(of),(built),(in),(patterns),(and),
(parsers),(as),(well),(as),(a),(novel),(pattern),(l
anguage),(called),(text),(constraints),(Tools),(LAP
IS),(includes),(a),(collection),(of),(text),(proces
sing),(tools),(such),(as),(filtering),(sorting),(an
d),(replacing),(Commands),(You),(can),(run),(Tcl),(
commands),(and),(external),(programs),(from),(LAPIS
),(and),(automate),(interactions),(with),(web),(sit
es),(Simultaneous),(Editing),(LAPIS),(is),(also),(a
),(text),(editor),(After),(selecting),(a),(group),(
of),(text),(regions),(you),(can),(edit),(all),(the)
,(regions),(simultaneously),(letting),(LAPIS),(auto
matically),(generalize),(your),(edits),(across),(al
l),(the),(selected),(regions)
```

Figure 8.3: Using copy and paste to extract a selection. (a) The selection (`Word`) is copied to the clipboard. (b) After making a new document with File/New, the clipboard is pasted, leaving an insertion point after each pasted region. (c) The user deletes the linebreaks that were inserted by default, and types `),(` instead. (d) The user fixes up the boundary cases, before the first and after the last extracted region.

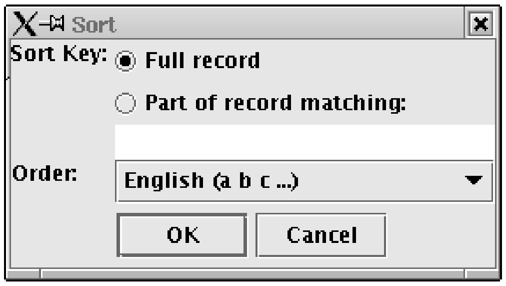


Figure 8.4: The Sort dialog.

8.1.2 Sort

The Sort command sorts the regions in the current selection. The selection is sorted in place, not extracted, so all text outside the selection is left unchanged. The output is a new document in which the selected regions are sorted, but which is otherwise identical to the input document. Some examples of the Sort command are shown in Figure 8.5.

The Sort menu command pops up a dialog box (Figure 8.4), which offers two options. The *sort key* specifies the part of each region that is compared to determine how the regions are sorted. By default, the entire region is used as the sort key. If only part of the region should be used as the sort key, the user must give a TC pattern describing that part. The sort key for a region is then determined by concatenating all matches to the key pattern found in the region. If a region contains no matches to the sort key pattern, its sort key is the empty string.

The *sort order* specifies the collation order for sort keys. The following sort orders are available:

- **<Local Language> (a b c ...)** sorts the keys lexicographically in ascending order, case-insensitively, using the collation order for the current locale reported by the Java runtime. For users of the US locale, for example, this option reads English. For users of the Spanish locale, this option reads Spanish, and sorts accented characters in the conventional Spanish collation order.

- **Reverse <Local Language> (z y x ...)** sorts the keys lexicographically in descending order, case-insensitively.

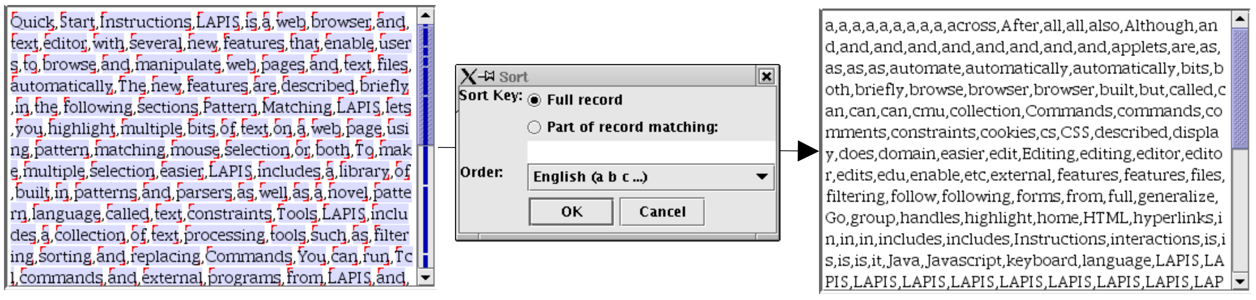
- **Numeric (1 2 3 ...)** and **Reverse Numeric (9 8 7 ...)** sort the keys as if they were numbers. Regions which do not start with a recognizable number are treated as 0.

- **Unicode (A B C ... a b c ...)** and **Reverse Unicode (z y x ... Z Y X ...)** sort the keys lexicographically and case-sensitively by comparing Unicode values.

- **Random** applies a random permutation to the selected regions. The content of the sort keys is irrelevant to this sort order.

The default sort order is <Local Language>, e.g., English.

The sort orders in LAPIS were chosen to cover the common cases with a few simple choices. Not all possible orderings are available. For example, no case-sensitive local-language sort order



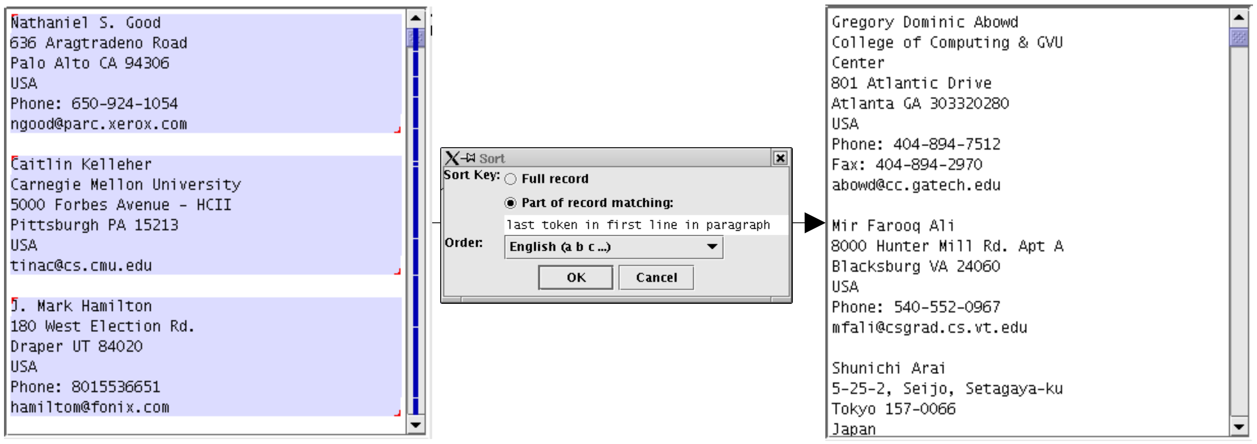
```
Quick,Start,Instructions,LAPIS,is,a,web,browser,and,
text,editor,with,several,new,features,that,enable,user
s,to,browse,and,manipulate,web,pages,and,text,files,
automatically,The,new,features,are,described,briefly
,in,the,following,sections,Pattern,Matching,LAPIS,lets
,you,highlight,multiple,bits,of,text,on,a,web,page,usi
ng,pattern,matching,mouse,selection,or,both,To,mak
e,multiple,selection,easier,LAPIS,includes,a,library,of
,built,in,patterns,and,parsers,as,well,as,a,novel,patte
rn,language,called,text,constraints,Tools,LAPIS,inclu
des,a,collection,of,text,processing,tools,such,as,filter
ing,sorting,and,replacing,Commands,You,can,run,Tc
l,commands,and,external,programs,from,LAPIS,and,
```

Sort Key: [✓ Full record] Part of record matching:

Order: English (a b c ...)

```
a,a,a,a,a,a,a,a,a,across,After,all,all,also,Although,an
d,and,and,and,and,and,and,and,and,applets,are,as,
as,as,as,automate,automatically,automatically,bits,b
oth,briefly,browse,browser,browser,built,but,called,c
an,can,can,cmu,collection,Commands,commands,co
mments,constraints,cookies,cs,CSS,described,displa
y,does,domain,easier,edit,Editing,editing,editor,edito
r,edits,edu,enable,etc,external,features,features,files,
filtering,follow,following,forms,from,full,generalize,
Go,group,handles,highlight,home,HTML,hyperlinks,i
n,in,in,includes,includes,Instructions,interactions,is,i
s,is,is,it,Java,Javascript,keyboard,language,LAPIS,LA
PIS,LAPIS,LAPIS,LAPIS,LAPIS,LAPIS,LAPIS,LAPIS,LAP
```

(a) sorting words



```
Nathaniel S. Good
636 Aragtradeno Road
Palo Alto CA 94306
USA
Phone: 650-924-1054
ngood@parc.xerox.com

Caitlin Kelleher
Carnegie Mellon University
5000 Forbes Avenue - HCII
Pittsburgh PA 15213
USA
tinac@cs.cmu.edu

J. Mark Hamilton
180 West Election Rd.
Draper UT 84020
USA
Phone: 8015536651
hamiltom@fonix.com
```

Sort Key: Full record [✓ Part of record matching:] last token in first line in paragraph

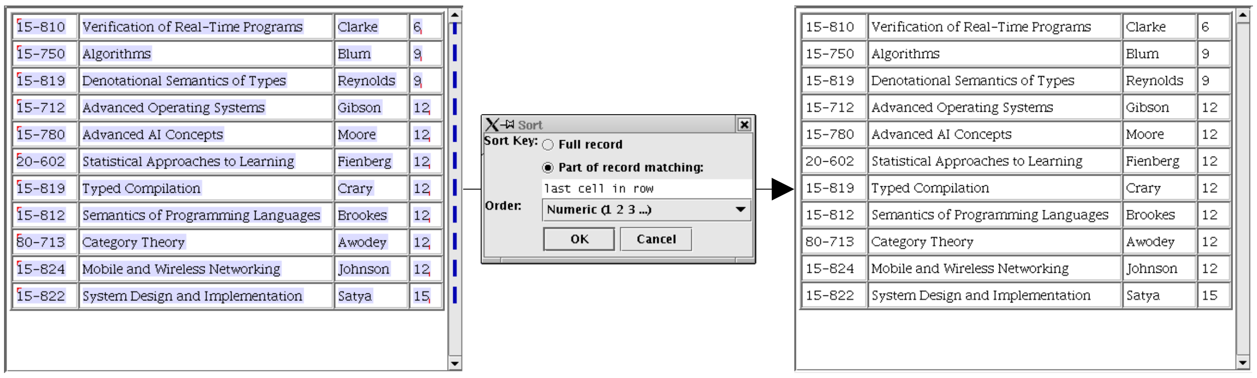
Order: English (a b c ...)

```
Gregory Dominic Abowd
College of Computing & GVU
Center
801 Atlantic Drive
Atlanta GA 303320280
USA
Phone: 404-894-7512
Fax: 404-894-2970
abowd@cc.gatech.edu

Mir Farooq Ali
8000 Hunter Mill Rd. Apt A
Blacksburg VA 24060
USA
Phone: 540-552-0967
mfali@csgrad.cs.vt.edu

Shunichi Arai
5-25-2, Seijo, Setagaya-ku
Tokyo 157-0066
Japan
```

(b) sorting addresses by last name



15-810	Verification of Real-Time Programs	Clarke	6
15-750	Algorithms	Blum	9
15-819	Denotational Semantics of Types	Reynolds	9
15-712	Advanced Operating Systems	Gibson	12
15-780	Advanced AI Concepts	Moore	12
20-602	Statistical Approaches to Learning	Fienberg	12
15-819	Typed Compilation	Crary	12
15-812	Semantics of Programming Languages	Brookes	12
80-713	Category Theory	Awodey	12
15-824	Mobile and Wireless Networking	Johnson	12
15-822	System Design and Implementation	Satya	15

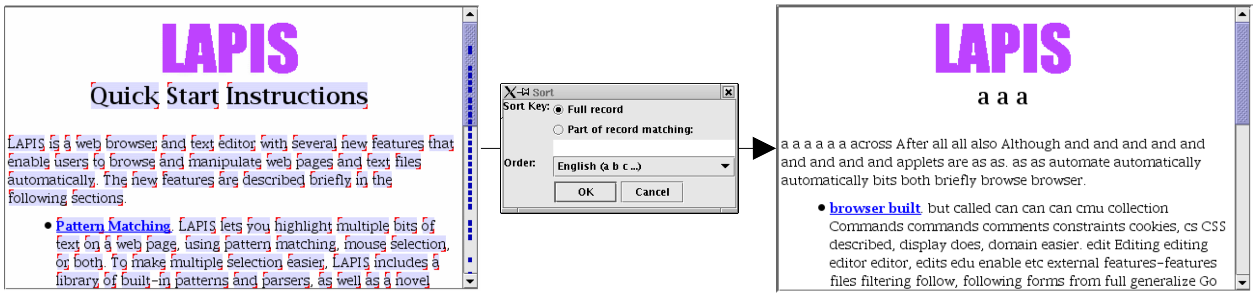
Sort Key: Full record [✓ Part of record matching:] last cell in row

Order: Numeric (1 2 3 ...)

15-810	Verification of Real-Time Programs	Clarke	6
15-750	Algorithms	Blum	9
15-819	Denotational Semantics of Types	Reynolds	9
15-712	Advanced Operating Systems	Gibson	12
15-780	Advanced AI Concepts	Moore	12
20-602	Statistical Approaches to Learning	Fienberg	12
15-819	Typed Compilation	Crary	12
15-812	Semantics of Programming Languages	Brookes	12
80-713	Category Theory	Awodey	12
15-824	Mobile and Wireless Networking	Johnson	12
15-822	System Design and Implementation	Satya	15

(c) sorting courses by number of units

Figure 8.5: Examples of using Sort on various tasks.



LAPIS

Quick Start Instructions

LAPIS is a web browser and text editor with several new features that enable users to browse and manipulate web pages and text files automatically. The new features are described briefly in the following sections.

- **Pattern Matching**. LAPIS lets you highlight multiple bits of text on a web page, using pattern matching, mouse selection, or both. To make multiple selection easier, LAPIS includes a library of built-in patterns and parsers, as well as a novel

Sort

Sort Key: [✓ Full record] Part of record matching:

Order: English (a b c ...)

LAPIS

a a a

a a a a a a across After all all also Although and and and and and and and and and applets are as as. as as automate automatically automatically bits both briefly browse browser.

- **browser built**. but called can can can cmu collection Commands commands comments constraints cookies, cs CSS described, display does, domain easier. edit Editing editing editor editor, edits edu enable etc external features-features files filtering follow, following forms from full generalize Go

Figure 8.6: Sorting the words in a document produces gibberish. (Note that the big purple "LAPIS" was not sorted with the other words because it is an image, not text.)

is provided. If case-sensitivity is desired, the user must choose Unicode order. Other sort orders would also be useful, including dates, times, and sort orders that ignore whitespace and punctuation when comparing sort keys. The most general solution would let the user specify a custom sort order with a comparison function, like the standard C function `qsort` [KR88] or the Perl function `sort` [WCS96].

The script command for Sort has the following syntax:

```
sort pattern
     [-by keypattern]
     [-order [reverse] dictionary|numeric|unicode|random]
```

Multiple sort keys can be specified with multiple `-by` arguments. Sort keys are compared in the order that the `-by` arguments appear, so the first `-by` argument specifies the primary key. If two regions have the same primary key, the next sort key is compared, and so on. The default sort order for each key is `dictionary`, the local language. To change the sort order for a key, the `-by` option must be followed by a `-order` option. For example, the following command might sort a list of cars primarily by year (in reverse numeric order), then by make:

```
sort Car -by Year -order reverse numeric -by Make
```

The interactive Sort dialog may support multiple sort keys in a future version of LAPIS.

Sorting arbitrary regions in place is powerful, but also potentially dangerous. For example, it is easy to reduce a document to gibberish by sorting its words (Figure 8.6). More seriously, a user intending to sort a list of addresses by zip code might inadvertently select and sort the zip codes, instead of selecting the addresses and using the zip code as a sort key. Sorting just the zip codes in place would leave the addresses in the same order but reassign a different zip code to each address, resulting in corrupted data. This problem is not unique to LAPIS. In one case reported in the Boston Globe and `comp.risks`, a teacher sorted only one column in a spreadsheet of student grades, which scrambled the grades as a result [Lut00].

The Unix `sort` command is roughly equivalent to the LAPIS command `sort Line -order unicode`.

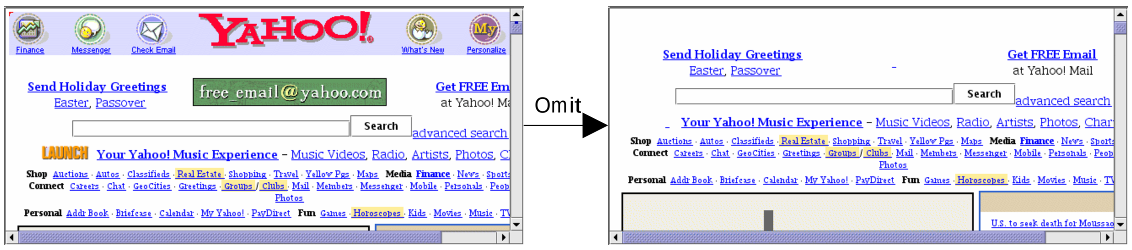


Figure 8.7: Images can be deleted from a web page by selecting them (with the `Image` pattern) and applying the Omit command.

8.1.3 Keep and Omit

The next two commands, Keep and Omit, are closely related, so this section describes both. The main use of these commands is *filtering* — removing undesired information from a document. Unlike the Extract command, the Keep and Omit commands filter information in place, preserving context outside the regions being filtered.

Omit is the easier of the two commands to understand. The Omit command produces a new document with the current selection deleted. For example, Figure 8.7 shows the result of omitting the images from a web page.

The Omit command requires the user to select the text that they *don't* want to see. Often, however, it makes more sense to select what you *do* want to see and filter everything else out. The Keep command is designed for this purpose. Keep does not simply delete all the text outside the selection, however.. Deleting all unselected text may delete important context, like table headings or data labels, which may be needed to understand the data that remains. Instead, Keep uses a simple inference technique to determine the set of *records* that the user is trying to filter. Then Keep deletes only the unselected records, preserving the selected ones and any context outside the records. Figure 8.8(b) shows an example of Keep applied to a web page.

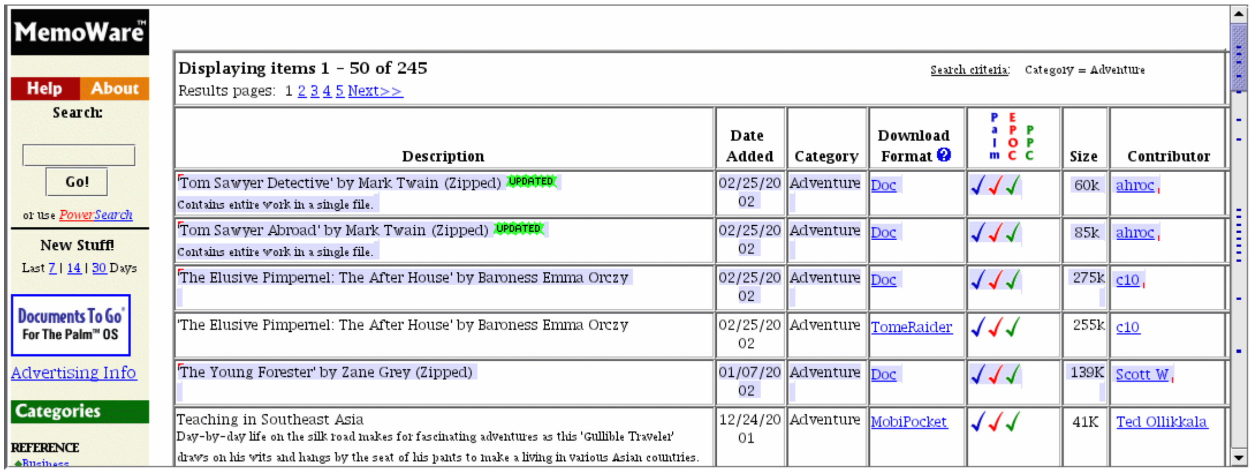
The record set used by Keep is inferred automatically by stripping constraints away from the TC pattern that created the current selection. (If the selection was created by the mouse, the appropriate record set cannot be inferred by this technique, and the Keep command is disabled.) As long as the pattern has the form *`A op B`* where *`op`* is either a relational operator, `and`, or `not`, the constraint *`op B`* is stripped away. Several examples of this reduction are shown below.

```
Word starting "s"         --> Word
     ending "e"

either Link or Image      --> either Link or Image
   just after Heading

from "<" to ">"           --> from "<" to ">"
   not containing "img"
```

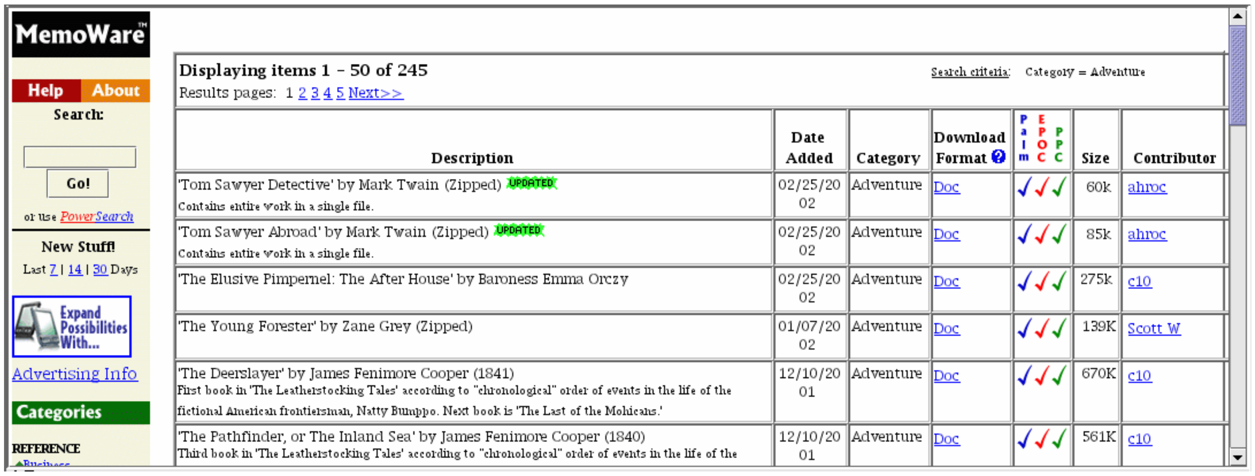
This inference technique is very simple, but it has the advantage of being more predictable than machine learning techniques like the selection inference algorithms described in Chapter 9. Often the record set is very simple, like `Line` or `Row` or `Method`. Sometimes, however, the record set



Displaying items 1 - 50 of 245 Search criteria: Category = Adventure
Results pages: 1 2 3 4 5 Next>>

Description	Date Added	Category	Download Format	Palm / EPOC / PPC	Size	Contributor
'Tom Sawyer Detective' by Mark Twain (Zipped) UPDATED Contains entire work in a single file.	02/25/2002	Adventure	Doc	✓✓✓	60k	ahroc
'Tom Sawyer Abroad' by Mark Twain (Zipped) UPDATED Contains entire work in a single file.	02/25/2002	Adventure	Doc	✓✓✓	85k	ahroc
'The Elusive Pimpernel: The After House' by Baroness Emma Orczy	02/25/2002	Adventure	Doc	✓✓✓	275k	c10
'The Elusive Pimpernel: The After House' by Baroness Emma Orczy	02/25/2002	Adventure	TomeRaider	✓✓✓	255k	c10
'The Young Forester' by Zane Grey (Zipped)	01/07/2002	Adventure	Doc	✓✓✓	139K	Scott W
Teaching in Southeast Asia Day-by-day life on the silk road makes for fascinating adventures as this 'Gullible Traveler' draws on his wits and hangs by the seat of his pants to make a living in various Asian countries.	12/24/2001	Adventure	MobiPocket	✓✓✓	41K	Ted Ollikkala

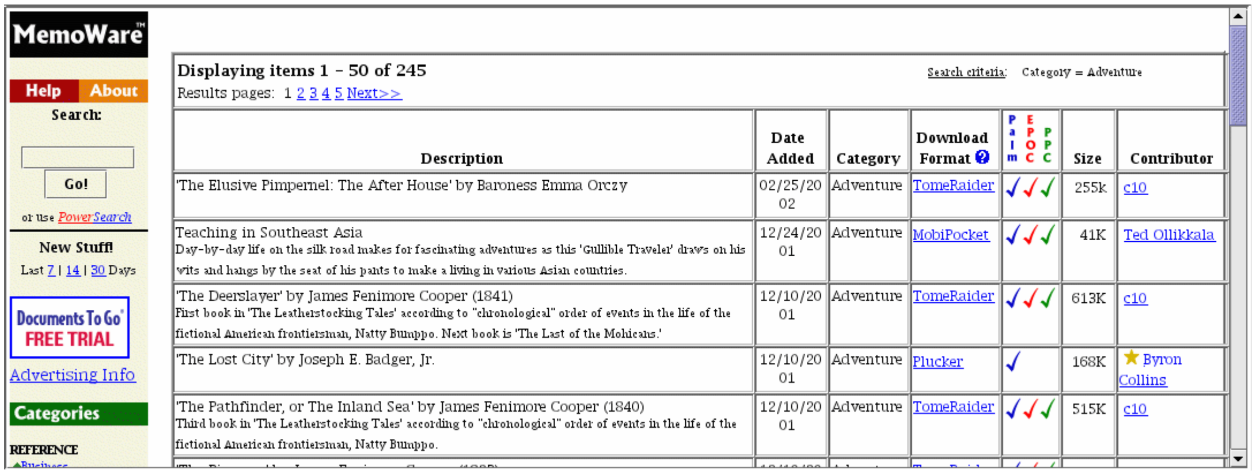
(a) Select `Book contains "Doc"`



Displaying items 1 - 50 of 245 Search criteria: Category = Adventure
Results pages: 1 2 3 4 5 Next>>

Description	Date Added	Category	Download Format	Palm / EPOC / PPC	Size	Contributor
'Tom Sawyer Detective' by Mark Twain (Zipped) UPDATED Contains entire work in a single file.	02/25/2002	Adventure	Doc	✓✓✓	60k	ahroc
'Tom Sawyer Abroad' by Mark Twain (Zipped) UPDATED Contains entire work in a single file.	02/25/2002	Adventure	Doc	✓✓✓	85k	ahroc
'The Elusive Pimpernel: The After House' by Baroness Emma Orczy	02/25/2002	Adventure	Doc	✓✓✓	275k	c10
'The Young Forester' by Zane Grey (Zipped)	01/07/2002	Adventure	Doc	✓✓✓	139K	Scott W
'The Deerslayer' by James Fenimore Cooper (1841) First book in 'The Leatherstocking Tales' according to "chronological" order of events in the life of the fictional American frontiersman, Natty Bumppo. Next book is 'The Last of the Mohicans.'	12/10/2001	Adventure	Doc	✓✓✓	670K	c10
'The Pathfinder, or The Inland Sea' by James Fenimore Cooper (1840) Third book in 'The Leatherstocking Tales' according to "chronological" order of events in the life of the	12/10/2001	Adventure	Doc	✓✓✓	561K	c10

(b) Invoke Keep command on (a)



Displaying items 1 - 50 of 245 Search criteria: Category = Adventure
Results pages: 1 2 3 4 5 Next>>

Description	Date Added	Category	Download Format	Palm / EPOC / PPC	Size	Contributor
'The Elusive Pimpernel: The After House' by Baroness Emma Orczy	02/25/2002	Adventure	TomeRaider	✓✓✓	255k	c10
Teaching in Southeast Asia Day-by-day life on the silk road makes for fascinating adventures as this 'Gullible Traveler' draws on his wits and hangs by the seat of his pants to make a living in various Asian countries.	12/24/2001	Adventure	MobiPocket	✓✓✓	41K	Ted Ollikkala
'The Deerslayer' by James Fenimore Cooper (1841) First book in 'The Leatherstocking Tales' according to "chronological" order of events in the life of the fictional American frontiersman, Natty Bumppo. Next book is 'The Last of the Mohicans.'	12/10/2001	Adventure	TomeRaider	✓✓✓	613K	c10
'The Lost City' by Joseph E. Badger, Jr.	12/10/2001	Adventure	Plucker	✓	168K	★ Byron Collins
'The Pathfinder, or The Inland Sea' by James Fenimore Cooper (1840) Third book in 'The Leatherstocking Tales' according to "chronological" order of events in the life of the fictional American frontiersman, Natty Bumppo.	12/10/2001	Adventure	TomeRaider	✓✓✓	515K	c10

(c) Invoke Omit command on (a)

Figure 8.8: Keep and Omit are complementary. Starting with the selection shown in (a), the Keep command keeps the selected books and deletes the rest (b), while the Omit command deletes the selected books and keeps the rest (c). `Book` is defined as `Row anywhere after "Description"`.

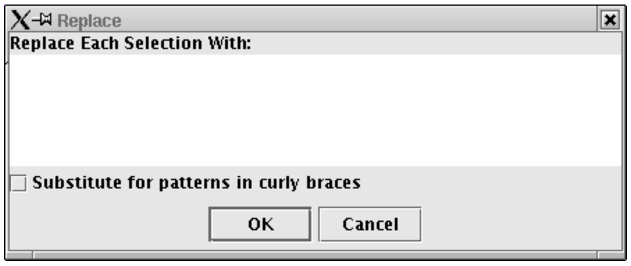

Replace
Replace Each Selection With:
Substitute for patterns in curly braces
OK
Cancel

Figure 8.9: The Replace dialog.

needs to be described by a complex pattern. In this case, the user's selection pattern may include some constraints that define the record set and other constraints that select a particular subset of records. The Keep command cannot automatically discriminate among these constraints, so the best way to use Keep is to first define a record pattern and assign it a name, and then use that name in subsequent selection patterns.

When the selection pattern follows the form `A op B`, Keep and Omit are complementary operations. For example, applying Keep to the selection `Link containing EmailAddress` produces the same result as applying Omit to the selection `Link not containing EmailAddress`, and vice versa. Figure 8.8 contrasts the behavior of Keep and Omit on a selection.

The Keep and Omit menu commands have corresponding script commands:

```
keep pattern [-outof recordpattern]
omit pattern
```

By default, the `keep` command infers its record set using the same technique as the Keep menu command. The user can override the inference with the `-outof` option, which specifies the record set using another pattern. The `omit` command has no need for a record set.

No conventional Unix tools are precisely analogous to Keep and Omit. The closest equivalent might be `grep -v regexp`, which extracts lines that fail to match `regexp`. The LAPIS equivalent for this command would be, roughly,

```
omit {Line containing /regexp/}
```

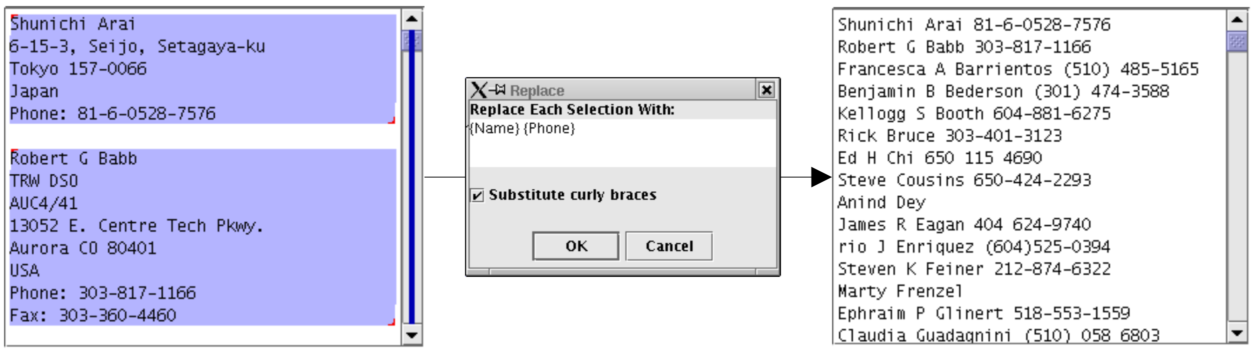
or equivalently,

```
extract {Line not containing /regexp/}
```

8.1.4 Replace

The Replace command replaces every selected region with another string. Invoking Replace pops up a dialog box that prompts the user for replacement text (Figure 8.9). If the user types some text and presses OK, then the Replace command generates a new document in which every selection is replaced by the replacement text. The Replace dialog uses a multi-line text widget, so the replacement text may include embedded newlines.

The simplest kind of Replace substitutes the same string for every selection. If the user checks the "Substitute for patterns in curly braces" checkbox, however, then the replacement text is treated



```
Shunichi Arai
6-15-3, Seijo, Setagaya-ku
Tokyo 157-0066
Japan
Phone: 81-6-0528-7576

Robert G Babb
TRW DSO
AUC4/41
13052 E. Centre Tech Pkwy.
Aurora CO 80401
USA
Phone: 303-817-1166
Fax: 303-360-4460
```

```
Shunichi Arai 81-6-0528-7576
Robert G Babb 303-817-1166
Francesca A Barrientos (510) 485-5165
Benjamin B Bederson (301) 474-3588
Kellogg S Booth 604-881-6275
Rick Bruce 303-401-3123
Ed H Chi 650 115 4690
Steve Cousins 650-424-2293
Anind Dey
James R Eagan 404 624-9740
rio J Enriquez (604)525-0394
Steven K Feiner 212-874-6322
Marty Frenzel
Ephraim P Glinert 518-553-1559
Claudia Guadagnini (510) 058 6803
```

Figure 8.10: Replacing selections with a template. Every region in the selection on the left (made by the pattern `Paragraph`) is replaced by a template with two slots — one for the person's name (`Name`, defined as `first Line in Paragraph`) and the other for the person's phone number (`Phone`, defined as `from end of "Phone:" to start of Linebreak`). When a template pattern fails to match, e.g., when there isn't a phone number, that slot in the template is left blank.

as a *template*. The template may contain one or more slots for substitution, each denoted by a TC pattern surrounded by curly braces. Figure 8.10 shows an example of a template substitution. To perform the replacement, each selected region is searched for matches to the substitution patterns, and the text of the matches are filled into the template before replacing the selected region. If more than one region matches the substitution pattern, then the multiple matches are simply concatenated together. If no regions match the substitution pattern, then the slot is filled by the empty string. The substitution pattern `{all}` can be used to match the entire original region, so that the replacement template can specify text to insert before or after it. Literal curly braces in the replacement template must be escaped with backslashes, and backslashes must also be escaped.

The Replace dialog is clunky and largely unnecessary now that LAPIS supports multiple-selection editing (Section 7.5). Multiple-selection editing offers more interactive ways to do both kinds of replacement. To replace every selection with the same string, the user can simply type the string into the browser pane. By the rules of multiple-selection editing, this has the effect of deleting all the selections and replacing them with the typed string. To replace every selection with a template, the most interactive technique is simultaneous editing (Chapter 9), which effectively infers substitution patterns from the user's examples. Editing only works in the plain text view, however, so the Replace command is still the best option for replacements in the HTML view.

The Replace menu command has a corresponding script command:

```
replace pattern replacement
```

where *`pattern`* describes the regions to be replaced and *`replacement`* is the replacement template.

The closest Unix analog to Replace is the `sed` command, which allows the user to specify substitutions of the form

```
s/regexp/template/
```

Here, `template` may refer to parenthesized subexpressions of `regexp` by number, `\0` through `\9`, or to the entire string matched by `regexp` using the special character `&`. The Replace template provides similar functionality, except that the template specifies its own patterns for substitution, rather than referring back to subexpressions of the original pattern. Perl [WCS96] and awk [AKW88] provide a substitution command similar to `sed`'s.

8.1.5 Calc

The last command on the Tools menu is Calc. The Calc command attempts to interpret the selections as numbers and compute some statistics on them. By default, Calc produces an output document displaying all the statistics:

```
count = <number of numeric selections>
sum = <sum of selections>
average = <sum / count>
min = <minimum value of selections>
max = <maximum value>
stddev = <standard deviation of selections>
```

Nonnumeric selections are ignored, so the count only includes the selections that are recognizable as numbers. Statistics are computed using Java `doubles`, which are 64-bit IEEE floating point values.

The Calc command has a corresponding script command:

```
calc pattern
     [-count]
     [-sum]
     [-average|-mean|-avg]
     [-min]
     [-max]
     [-stddev]
```

in which `pattern` describes the regions that should be processed. If no options are specified, then `calc` returns all the statistics in the format shown above. If one or more options is specified, then `calc` only returns the requested statistics, separated by spaces. For example, the command

```
calc {Number just after "Price:"} -min -max -mean
```

might return

```
400 850 525.5
```

No conventional Unix command corresponds to Calc.

8.1.6 Count

There is no Count command on the Tools menu, because the number of matches to a pattern is displayed automatically. Whenever the user runs a pattern or changes the selection with the mouse, LAPIS displays the number of regions currently selected in the pattern pane (Figure 7.8).

The script command `count` does the same thing:

```
count pattern
```

In scripts, the `count` command is most useful for testing whether a pattern has any matches.

The closest Unix analog for this command is `grep -c` *`regexp`*, which counts lines that match a pattern. Another similar Unix utility is `wc`, which displays the number of words and lines in the document. The same effect can be achieved by `count Word` and `count Line`.

8.2 The Browser Shell

Most interactive interpreters use a *typescript* interface, in which command prompts are interleaved with command output in the same window. Lisp, Tcl, Python, and Unix shells like `sh` and `csh` all follow this interaction model.

In contrast, LAPIS integrates its interpreter directly into the browsing window, a new idea called a *browser shell*. A browser shell is different from a typescript shell in three ways:

- **Commands are typed into the Command bar.** The Command bar in LAPIS fills a dual role. Like the Location box in other web browsers, it displays the URL of the web page or file in the browser pane. However, the user can also type script commands into it.

- **The browser pane acts as standard input and output for commands.** When a command is executed, it takes its input from the current document in the browser pane, which may be a file, a web page, or the output of a previous command. After executing, the command's output is sent back to the browser pane as a new page, where the user can view it as either plain text or rendered HTML, scroll through it, edit it, or apply further commands.

- **Executed commands appear in the browsing history.** The Forward and Back browsing buttons navigate through command output pages as well as web pages. When the browsing history is viewed as a list, both URLs and executed commands appear in the list. Portions of the history list can be extracted and saved as a script for later reuse.

The remainder of this chapter discusses the details and implications of the LAPIS browser shell.

8.3 URLs as Commands

Either a URL or a script command may be typed into the Command bar. LAPIS takes this integration to its logical conclusion, and simply defines URLs as commands in its dialect of Tcl. All of the following are valid commands in LAPIS:

```
http://www.yahoo.com/
ftp://ftp.cs.cmu.edu/afs/cs/project/amulet
file:/home/rcm
```

The result of a URL command is the page or file loaded from that URL. File or FTP URLs that refer to directories return a page listing the files in the directory.[3] Regarding URLs as commands has two advantages. First, it simplifies the implementation of the Command bar, so that it can treat everything typed into it as a script command. Discrimination between URLs and other script commands happens at a lower level. Second, and more important, URLs can be used directly in scripts. For example, here is a simple script that extracts the Pittsburgh weather forecast from Yahoo:

```
http://weather.yahoo.com/forecast/USPA1290_f.html
extract {Table just after "5 day forecast"}
```

URL commands are implemented using Tcl's unknown-command feature. When Tcl encounters a command name that doesn't exist, it invokes `unknown` instead, passing the unknown command's name and arguments. In LAPIS, `unknown` checks whether the unknown command can be parsed as a URL. If so, the URL is fetched and its contents are returned as the result of the command.

LAPIS also treats filenames as commands. Thus these are all commands:

```
/etc/passwd
../../index.html
fruits.txt
```

Like a URL, a filename command loads the file into LAPIS. Because filename commands are also implemented with the Tcl unknown-command feature, files with the same name as a Tcl command cannot be loaded this way unless they are explicitly disambiguated using the `file:` prefix. For example, to load a file named `extract`, which is ambiguous with the LAPIS built-in `extract` command, the file command must be

```
file:extract
```

Filename commands may also be ambiguous with external programs (Section 8.6), in which case the `file:` prefix can again be used to disambiguate.

Relative filenames and URLs are resolved relative to the current directory. By default, the current directory is the directory of the last file loaded. The user can also change the current directory explicitly using the Tcl `cd` command. The current directory is used not only for resolving relative filenames, but also to invoke external programs, and as the starting directory for the File/Open and File/Save dialog boxes.

[3]The format of this page is not well-defined, unfortunately. LAPIS uses the Java library to load URLs, and the Java library has changed the format of the listing returned by directory URLs from one release of Java to the next.

8.4 Self-Disclosure in the Command Bar

The Command bar is also used for *self-disclosure* [DE95]. Self-disclosure is a technique for helping users learn more about a powerful user interface, particularly about features which are otherwise hidden, such as scripting language commands and syntax. Self-disclosure has been used in other systems to teach about keyboard shortcuts (Lyx [Lyx02]) and script commands (Emacs [Sta81], AutoCAD [Aut02], Chart N' Art [DE95]). In LAPIS, self-disclosure is used to expose the user to both the scripting language and the pattern language.

The Location box in other web browsers is already used for a kind of self-disclosure, because it continually tracks the URL of the current web page. When the user clicks on a hyperlink to browse to another page, the Location box is updated to show the new page's URL. In a sense, the Location box is disclosing a command — the URL — that can be used to achieve the same effect as the user's graphical interaction — a click on a hyperlink. Self-disclosure is probably not as valuable for URLs as it is for other kinds of commands, since there is no consistent "URL language" across web sites that would help a user generate a URL from scratch. Nevertheless, users do take advantage of the disclosed URL for saving bookmarks or sending web pages to other people via email or instant messaging.

In keeping with the unification of URLs and other script commands, LAPIS not only discloses URLs in its Command bar, but also script commands. When the user invokes a text-processing command from the Tools menu, a script command that would have had the same effect is displayed in the Command bar. For example, suppose the user selects all the words in the document using the pattern `Word`, and then invokes the Extract command to extract them. Then the command bar would display

```
extract Word
```

More examples of disclosed commands are shown in Table 8.11.

The arguments for a disclosed command are determined as follows. If the selection was created by running a pattern (like `Word`), then that pattern is used as the argument for the command. If any part of the selection was made by the mouse, then `Selection` is used as the argument for the command. If the user provided additional arguments to the menu command using a dialog box, then the corresponding arguments are included in the disclosed script command.

Self-disclosure is used in other parts of LAPIS as well. For example, clicking on a named pattern in the library pane shows in the pattern pane how the name would be referenced in a TC pattern (Section 7.4.3). Also, when LAPIS infers selections from examples, it discloses a TC pattern corresponding to the inferred selection (Chapter 9).

8.5 Command Input and Output

By default, commands take their input from the *current document*, which is the document displayed in the browser pane. After a command runs, its result becomes the new current document, and the original document is added to the browsing history. As a result, a sequence of commands implicitly forms a pipeline, with the output of one command becoming the input for the next. For example, this command sequence picks a random word starting with "a" from the list of words in `/usr/dict/words`:

Figure 8.2(a)	`extract Word`
Figure 8.2(b)	`extract Sentence`
Figure 8.2(c)	`extract Link`
Figure 8.5(a)	`sort Word`
Figure 8.5(b)	`sort Paragraph -by {last token in first line in paragraph}`
Figure 8.5(c)	`sort Row -by {last cell in row} -order numeric`
Figure 8.7	`omit Image`
Figure 8.8(top)	`keep {Book contains "Doc"}`
Figure 8.8(bottom)	`omit {Book contains "Doc"}`
Figure 8.10	`replace Paragraph {{Name} {Phone}}`

Figure 8.11: Commands disclosed for earlier examples in this chapter.

```
file:/usr/dict/words
extract {Word starting "a"}
sort Word -order random
extract {first Word}
```

The input for a command can be redirected from a different source by passing it as the last argument to the command. For example, this command extracts the words starting “a” from the given string:

```
extract {Word starting "a"} "apple banana pear apricot"
```

The result is

```
apple
apricot
```

since the `extract` command uses linebreaks as its default terminators. All the script commands described in Section 8.1 take an optional last argument specifying the input string to process.

The output of a command can be redirected using Tcl subexpression syntax `[`*`expr`*`]`. For example,

```
set dictionary [file:/usr/dict/words]
```

stores the document loaded from `/usr/dict/words` into the Tcl variable `dictionary`. A pipeline can also be expressed more verbosely by stacking up subexpressions, as in:

```
sort Word -order random \
     [extract {Word starting "a"} \
              [file:/usr/dict/words]]
```

(The backslashes in this command are Tcl syntax allowing a long command to be continued to the next line.)

When the LAPIS interpreter evaluates a nested context — a subexpression in square brackets — it automatically pushes the current document onto a stack, restoring it after evaluating the nested context. This behavior allows pipelines to be encapsulated in Tcl procedures and used in other pipelines. For example, the weather-fetching script shown earlier might be wrapped in a procedure called `get-weather`:

```
proc get-weather {} {
    http://weather.yahoo.com/forecast/USPA1290_f.html
    extract {Table just after "5 day forecast"}
}
```

and then used in a pipeline that inserts the weather forecast into the user's home page:

```
http://www.cs.cmu.edu/~rcm/personal-homepage.html
replace {"Weather Forecast"} [get-weather]
```

Note that the result of a Tcl procedure is the result of the last command in its body, so `get-weather` returns the weather forecast without needing an explicit `return` command. This example assumes that the user's home page contains the words "Weather Forecast" somewhere in it, as a placeholder for the weather forecast.

8.6 External Programs

LAPIS can also run an external command-line program from the Command bar. Like URL evaluation, this feature is implemented using Tcl's unknown-command facility. If a command name is not found as a built-in Tcl command or user-defined procedure, and cannot be parsed as a URL, then LAPIS searches for an external program by that name. For example, typing the command

```
ls
```

would display a listing of the files in the current directory (at least on systems where `ls` is available). If an external program has the same name as a Tcl command, then the Tcl command takes precedence. Thus, `sort` refers to the LAPIS sort command, not the system sort program. The user can force the external program to run instead by using the `exec:` prefix. Thus

```
exec:sort
```

runs the external `sort` program instead of the LAPIS `sort`. The `exec:` prefix is intended to be analogous to other URL protocol prefixes like `http:` and `file:`.

Like other commands, an external program automatically takes its input from the current document, and its output becomes the new current document in the browser pane. Thus, external programs can participate in pipelines. For example, if the user types (on BSD-style Unix)

```
ps -aux
```

then the browser displays a list of running processes. If the next command is

```
grep xclock
```

then the process listing is filtered to display only those lines containing `xclock`. (The same effect could be achieved by the LAPIS command `extract {Line containing "xclock"}`.)

To make this work with legacy programs like `ps` and `grep`, the external program is invoked in a subprocess with its input and output redirected. Standard input is redirected from the current page of the browser (passing the HTML source if the current page is a web page). Standard output is redirected to a new document displayed in the browser pane, which is displayed incrementally as the program writes its output. Standard error is redirected to a special subframe in the browser pane, in order to separate it from the standard output. Figure 8.12 shows the resulting output for two different invocations of `ls`. The standard error pane normally remains hidden until an external program prints to it.

Program output may be parsed and manipulated like any other page in LAPIS. For example, `ps -aux` displays information about running processes in a tabular form:

```
USER    PID %CPU %MEM SIZE RSS TTY...
bin     160  0.0  0.4  752 320  ? ...
daemon  194  0.0  0.6  784 404  ? ...
rcm     294  0.0  1.0 1196 660  ? ...
```

This output can be described by TC patterns entered interactively in the pattern pane.

```
Process is Line not 1st Line
User is Word starting Process
PID is Number just after User
```

In a script, TC patterns can be run by the `parse` command, which is handy for defining a group of named patterns for use in subsequent commands. The `parse` command takes one argument, which is a TC expression or group of TC expressions:

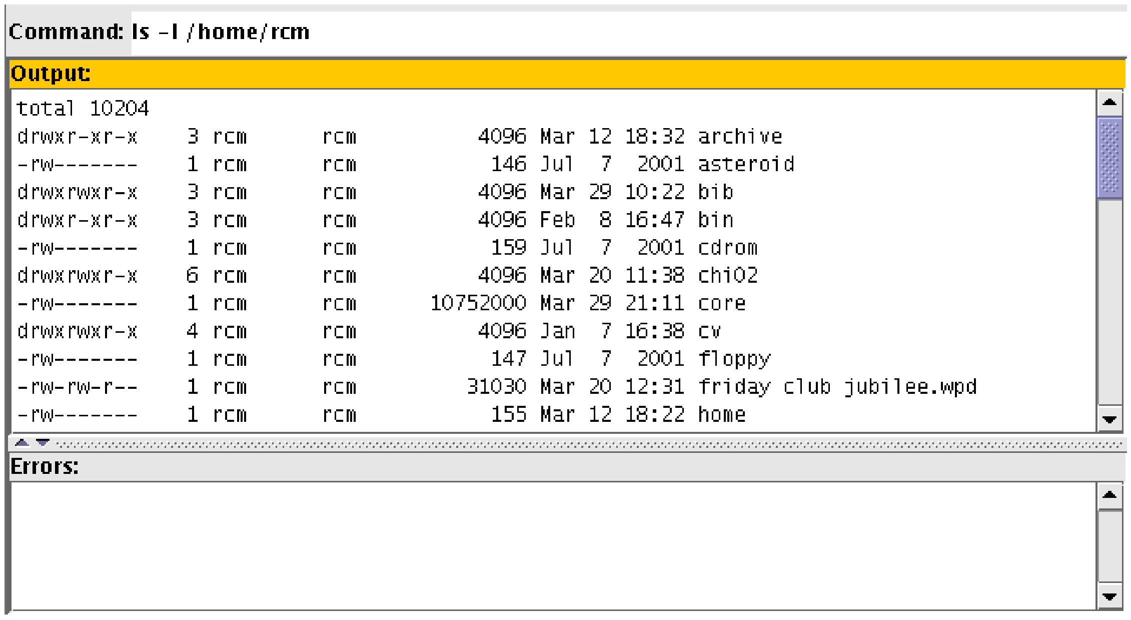
```
parse {
    Process is Line not 1st Line
    User is Word starting Process
    PID is Number just after User
}
```

These named patterns can then be used with LAPIS commands that search and manipulate `ps` output:

```
# sort processes by PID
sort Process -by PID -order numeric

# display only xterm processes
keep {Process contains "xterm"}

# kill all xterms
eval kill [extract {PID in Process contains "xterm"} -as tcl]
```



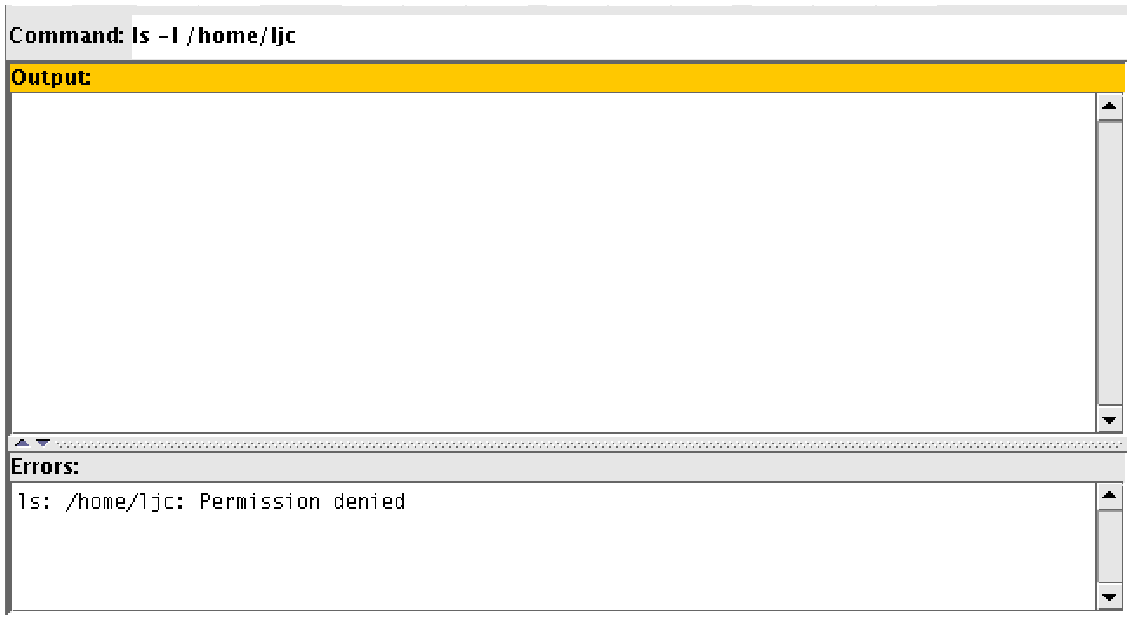
Command: ls -l /home/rcm

Output:

```
total 10204
drwxr-xr-x    3 rcm      rcm          4096 Mar 12 18:32 archive
-rw-------    1 rcm      rcm           146 Jul  7  2001 asteroid
drwxrwxr-x    3 rcm      rcm          4096 Mar 29 10:22 bib
drwxr-xr-x    3 rcm      rcm          4096 Feb  8 16:47 bin
-rw-------    1 rcm      rcm           159 Jul  7  2001 cdrom
drwxrwxr-x    6 rcm      rcm          4096 Mar 20 11:38 chi02
-rw-------    1 rcm      rcm      10752000 Mar 29 21:11 core
drwxrwxr-x    4 rcm      rcm          4096 Jan  7 16:38 cv
-rw-------    1 rcm      rcm           147 Jul  7  2001 floppy
-rw-rw-r--    1 rcm      rcm         31030 Mar 20 12:31 friday club jubilee.wpd
-rw-------    1 rcm      rcm           155 Mar 12 18:22 home
```

Errors:

(a)



Command: ls -l /home/ljc

Output:

Errors:

```
ls: /home/ljc: Permission denied
```

(b)

Figure 8.12: An external program's standard output and standard error streams are displayed in separate panes. (a) a successful command that prints only to standard output; (b) an unsuccessful command that prints an error message to standard error.

Note that `eval` is needed in the last example because `extract` may return a list of process IDs. Without `eval`, this list would be passed to `kill` as a single argument with embedded spaces; with `eval`, the list is exploded into separate arguments.

Normally, standard output becomes the current document for subsequent patterns and commands. The user can choose to process the standard error instead by clicking on the standard error pane to make it active. The same switch can be made in a script using the command

```
property stderr
```

which returns the standard error property of the current document. The `property` command accesses document metadata, explained in more detail below (Section 8.13).

8.7 The Browser Shell as a System Command Prompt

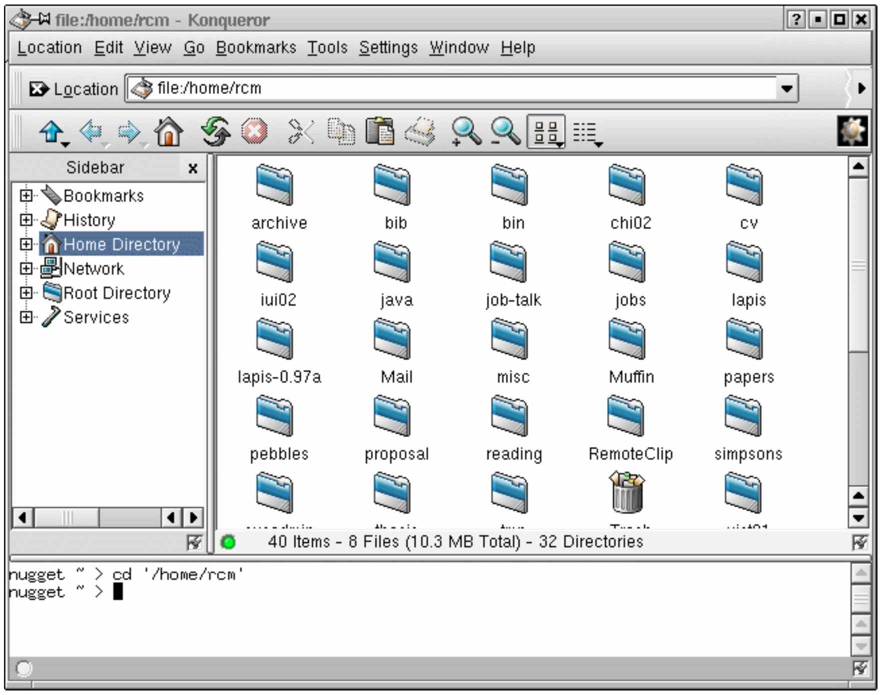
Since the LAPIS browser shell can invoke external programs, it is worth comparing it to the command prompt already available in operating systems like Unix, Windows, and Mac OS X. In a sense, the browser shell continues a trend which has seen the web browser become more and more tightly integrated into the desktop interface. Modern web browsers like Microsoft Internet Explorer and KDE Konqueror [KDE02] already include file management among the browser's responsibilities. Integrating the system command prompt is another step along the same path, a step which makes some sense because file management and command execution are often intertwined. Recognizing this fact, Konqueror also integrates a command prompt, which appears as a typescript shell in an optional pane at the bottom of the window (Figure 8.13).

The LAPIS browser shell interface behaves differently from a traditional typescript shell like Konqueror's, however. Whereas a typescript shell interleaves commands with program output in the same window, a browser shell separates the command prompt from program output. The browser shell also automatically redirects program input from the current browser page, and automatically sends program output to a new browser page.

One effect of these differences is on scrolling. In a typescript interface, long output may scroll out of the window. To view the start of the output, the user must either scroll back, or else rerun the command with output redirected to a command like `more` or `head`. The browser shell, by contrast, initially displays the *first* windowful of output, rather than the *last*, reducing the need for scrolling. When output is less than a windowful, a typescript shell can become cluttered by outputs of several commands, forcing the user to scan for the start of the latest output. The browser shell displays each program output on a new, blank page. In essence, the overall effect of the browser shell is like automatically redirecting the output of every command to `more`.

Unlike `more`, however, the browser shell's display is not ephemeral. The displayed output can be passed as input to another command, which allows pipelines to be assembled more fluidly than in the typescript interface. Developing a complicated Unix pipeline, such as `ps ax | grep xclock | cut -d ' ' -f 1`, is often an incremental process. In typescript interfaces, where input redirection must be specified explicitly, this process typically takes one of two forms:

- Repeated execution: run `A` and view the output; then run `A|B` and view the output; then (if `B` turned out wrong) run `A|B′` and view the output; etc. This strategy fails if any of the commands run slowly or have side-effects.



```
nugget ~ > cd '/home/rcm'
nugget ~ > 
```

Figure 8.13: The Konqueror web browser/file manager [KDE02] can include a terminal pane (at the bottom of the window) for executing commands.

- Temporary files: run `A > t1` and examine `t1`; then run `B` < `t1` > `t2` and examine `t2`; then (if `B` was wrong) run `B′` < `t1` > `t2`, etc.

The browser shell offers a third alternative: run `A` and view the output; then run `B` (which automatically receives its input from `A`) and view the output; then press Back (because `B` was wrong) and run `B′` instead. The browser shell displays each intermediate result of the pipeline while serving as automatic temporary storage.

Automatic input redirection makes constructing a pipeline very fluid, but it is inappropriate for programs that use standard input for interacting with the user, such as `passwd`. Such programs cannot be run directly in LAPIS. One solution is to pop up a terminal emulator in a different window, e.g.:

```
xterm -e passwd
```

In the future, it might be useful to embed a terminal emulator in LAPIS.

Another problem with the browser shell model is the linear nature of the browsing history. If the user runs `A`, backs up, and then runs `A′`, the output of `A` disappears from the browsing history. To solve this problem, LAPIS lets the user duplicate the browser window, including its history, so that one window preserves the original history while the other window is used to backtrack. (Netscape's New Window command used to work similarly before version 4.0.) A more complex solution might extend the linear browsing history to a branching tree, an idea which has been explored in at least one research web browser [AS95].

8.8 Page History vs. Command History

The browser shell actually has two distinct histories: a *page history* and a *command history*. Both histories are found in other web browsers, but only a command history is found in most typescript shells.

The *page history* consists of files, web pages, and command outputs. The current item in the page history is displayed in the browser pane. The user navigates through the page history using the Back and Forward buttons. The Go menu presents a listing of the page history, with each page identified by the URL or command that generated it. Clicking on a page in this list jumps directly to it. The page history in LAPIS corresponds to the browsing history in other web browsers.

The *command history* is a history of the commands that have been typed into the Command bar, made available as a drop-down menu under the Command bar. The user can pick an earlier command from the menu, edit it as necessary, and execute it again. The command history is analogous to the URL history in other web browsers, which is similarly rendered as drop-down menu under the Location box. Command history is also found in typescript shells, where it is usually accessed by pressing the up and down arrows on the keyboard.

As long as the user is simply typing commands, the page history and command history remain synchronized. Each time a command is entered, the command's output is added to the page history, and the command itself is added to the command history. For example, suppose the user has run two commands:

Command History	Page History
`http://www.cnn.com`	`http://www.cnn.com`
`extract Image`	`extract Image`

If the user clicks Back to undo the last command and run a different one, however, the two histories become unsynchronized. The new command is *added* to the command history, but its output *replaces* the output of the old `extract` in the page history:

Command History	Page History
`http://www.cnn.com`	`http://www.cnn.com`
`extract Image`	`extract {Image anywhere after "News"}`
`extract {Image anywhere after "News"}`	

The page history only records the sequence of commands that produced the current page. Any commands that were "undone" by Back are not included in the page history.[4]The command history records all commands that have been executed, whether or not they were retracted by Back.

The two histories also differ in size. By default, LAPIS only keeps the last 20 pages in the page history, since pages may be large and the current version of LAPIS stores all pages in RAM. If LAPIS were a full-fledged web browser, however, the page history would simply be part of the browser cache, with some pages in RAM and some on disk, and old pages evicted as necessary. The command history can be much longer than the page history with little impact on memory use. LAPIS limits the command history to 100 commands to keep the drop-down menu at a manageable length.

[4]Saying that Back "undoes" commands is not strictly accurate, of course — LAPIS does not undo command side-effects like setting Tcl variables or changes the filesystem.

8.9 Generating Scripts from the Page History

The page history can be used to generate a script from a command session. The Convert History to Script command on the Scripts menu pops up a script editing window with the commands from the page history loaded into it. The example above would generate the script

```
http://www.cnn.com
extract {Image anywhere after "News"}
```

Self-disclosure helps here, because it labels all pages in the page history with a script command, even pages generated by menu commands. Thus the user can *demonstrate* a script using menu commands, although no inference is done to generalize the script automatically. The user can explicitly request inference by entering selection guessing mode; more about this can be found in Chapter 9.

The generated script usually requires some editing, because it includes the entire page history of the current LAPIS window. Old commands irrelevant to the desired script must be deleted manually. The Demonstrate menu command described below (Section 8.11), solves this problem by starting a script demonstration in a fresh window with an empty page history.

Currently, LAPIS has no convenient mechanism for naming the generated script or making it persistent. At the moment, the typical solution is to wrap the script in a Tcl procedure, e.g.:

```
proc cnn-pictures {} {
    http://www.cnn.com
    extract {Image anywhere after "News"}
}
```

If the script is only needed temporarily, then it is sufficient to use the Run command in the script editing window to define `cnn-pictures` in the Tcl interpreter for the duration of the session. To make the script persistent, it must be added to the user's Tcl startup script (whose location is defined in the Preferences dialog). Ideally, this step should be much easier. For example, scripts saved to a particular directory might automatically become available as script commands.

Scripts can be added to the LAPIS toolbar using the `toolbar` command. For example,

```
toolbar CNN cnn-pictures
```

adds a button labeled CNN to the toolbar which invokes the `cnn-pictures` procedure defined above.

8.10 Web Automation

Since LAPIS is a web browser, it includes commands for simple kinds of automated web browsing, so that a script can navigate through a dynamic web site or online transaction. Web browsing has two basic actions: clicking on hyperlinks and submitting forms. Automated web browsing requires equivalent script commands for each of these actions.

Clicking on a static link, whose target doesn't change over time, has the same result as typing the URL into the Command bar. Thus the script command for clicking on a static link is simply the link's URL, such as:

```
http://weather.yahoo.com/
```

For some links, however, the target URL varies depending on when the page is viewed. Variable links are often found in online newspapers, for example, where links to top stories change from day to day. The `click` command can be used to click on a variable link by describing its location in the web page with a pattern. For example, this script clicks on the top story in Salon:

```
http://www.salon.com/
click {Link after Image in 3rd Cell in Row}
```

The `click` command throws a Tcl exception if its pattern does not match exactly one hyperlink on the page.

For entering data into forms, LAPIS provides the `enter` command. The `enter` command takes two arguments: the first is a pattern describing the form field to modify, and the second is the value to enter in the field. For text fields, this value is a string which is entered in the field directly. For menus or lists, the value is selected in the list. For radio buttons or checkboxes, the value should be "on" or "off" (or yes/no, true/false, or 1/0).

Forms are submitted either by the `submit` command, which uses the default submit button on the form, or by a `click` command specifying which button to click. For example, here is a script that searches Google for the LAPIS home page and clicks the I'm Feeling Lucky button to jump right to it:

```
http://www.google.com/
enter Textbox {LAPIS Lightweight Structure}
click {Button contains view source "I'm Feeling Lucky"}
```

(Note that `view source` is needed here because of a LAPIS bug: the button's label, "I'm Feeling Lucky", is not searchable in the rendered HTML view.)

The examples presented so far have been web-site-specific, but some browsing tasks are sufficiently uniform across web sites to be handled by a generic script. For example, the following script can log into many web sites, assuming the user's login name and password have been stored in the Tcl variables `id` and `password`:

```
enter {
  Textbox just after Text containing either "login"
                                         or "email"
                                         or "id"
                                         or "user"
} $id
enter {
  Textbox just after Text containing "password"
} $password
submit
```

8.11 Web Automation by Demonstration

Combining web automation with script generation from the history makes it possible to create web browsing scripts by demonstration. An early version of LAPIS included this capability [MM00], but it has since ceased to work because of changes in other parts of LAPIS. This section describes the feature as it originally worked, and explains why it has become broken.

To create a browsing script quickly, the user can *demonstrate* it by recording a browsing sequence. The demonstration begins with an arbitrary example page, the *input page*, showing in the browser. Invoking the Demonstrate Script menu command pops up a new browser window, in which the browsing demonstration will take place. A new window is created so that the browsing sequence can refer to the input page for parameters. Like any LAPIS browser window, the Demonstrate window records a browsing history: URLs visited and commands typed. Unlike a normal browser window, however, the Demonstrate window's history also records user events in form controls. For example, if the user types into a form field, the history records an equivalent `enter` command.

To fill in a form with text from the input page, the user can make a selection in the input page, then copy and paste it to a form field in the Demonstrate window. If the copied text was selected by searching for a pattern, then this action records the command `enter` *`field-name pattern`* in the history (where *`field-name`* is the name given to the form field in the HTML). If the copied data was selected manually, then the command `enter` *`field-name`* `Selection` is recorded in the history, so that when the script is run at a later time, `Selection` will return the user's selection at that time. More complex dependencies can be expressed by typing a Tcl command instead of pointing-and-clicking. For example, if a radio button should be selected only if the input page has certain features, then the user might type the command `if {[count` *`pattern`*`]} {click` *`field-name`*`}`.

Using Back and Forward, the user can revise the demonstration as necessary until the desired results are achieved. When the user is satisfied with the demonstration, the Demonstrate window is closed and the history is generated as a script.

Other systems have provided web automation by demonstration, notably LiveAgent [Kru97], which records a sequence of Netscape browsing actions as a macro. LAPIS demonstrations have two advantages over LiveAgent. First, the recorded transcript is represented by the browsing history, which is visible, familiar, and easy to navigate. A crucial part of making this work is that LAPIS inserts commands as well as URLs in the browsing history. Second, an experienced user can generalize the demonstration on the fly by typing commands at crucial points instead of pointing-and-clicking. Since LAPIS embeds a full scripting language, the resulting scripts can be significantly more expressive than recorded macros, without taking much more time to develop.

The Demonstrate feature is currently broken because the *`field-name`* patterns used to identify form fields are outdated. Earlier versions of LAPIS followed a model in which *all* parsers were run on a page automatically, so that the HTML parser could create document-specific identifiers like *`field-name`* for each form field found in a page. This model doesn't scale well as the library of patterns and parsers grows. Instead, the current version of LAPIS only runs parsers on demand, as described in Section 6.2.4. This model is far more efficient, but it means that all the named patterns in the library must be document-independent — parsers cannot generate document-specific named patterns on the fly. In the new model, instead of simply using *`field-name`*, Demonstrate would have to describe each form field as follows:

```
Control
  contains name-attr
             contains AttributeValue
                        contains view source "field-name"
```

A better solution would be to add parameterized named patterns to TC, so that one can define a parameterized pattern `Control(field-name)` as the pattern above and refer to named form fields conveniently. Fixing Demonstrate, and adding parameterized named patterns, are both left for future work.

8.12 Other Tcl Commands

Built-in Tcl commands can also be entered in the Command bar. For example, the `expr` command performs arithmetic computation:

```
expr 5*5+2
```

This command returns 27 (as a string, the only data type in Tcl).

Unlike other LAPIS commands and external programs, built-in Tcl commands do not automatically use the current document for input and output. To give these commands access to the current document, LAPIS provides the `doc` command. Writing `doc` with no arguments simply returns the current document, so

```
string length [doc]
```

returns the length of the current document.

Writing `doc` *`string`* sets the current document to *`string`*. For example:

```
doc "This is the new document"
```

Both forms of `doc` can be used in a single Tcl command. For example, if the current document consists of a single number, the following command doubles it and sets the current document to the result:

```
doc [expr 2 * [doc]]
```

When a Tcl command is used in a pipeline, `doc` must be used to read and write the current document. For example, here is a pipeline that extracts the words from the current document, converts them all to lowercase using the built-in Tcl command `string tolower`, and then sorts them:

```
extract Word
doc [string tolower [doc]]
sort Word
```

At first, the `doc` command may seem like an unnecessary complication. Why not just modify the Tcl interpreter so that *all* Tcl commands set the current document — not only LAPIS extension commands like `extract` and `sort`, but also built-in commands like `string`? Then the pipeline above might be written instead as:

```
# this example won’t work in LAPIS
extract Word
string tolower [doc]
sort Word
```

This approach does not entirely eliminate `doc`, because it is still needed to specify the current document as an argument to `string`. Worse, however, this approach turns out to be unworkable because *all* Tcl commands return a string, including control structures like `while`, `foreach`, and `proc`. These commands always return the empty string, but are otherwise indistinguishable from other Tcl commands.[5] If the result of every Tcl command became the new current document, then it would be impossible to use control structures in a pipeline without clearing the current document.

When entering commands interactively, however, preceding every native Tcl command with `doc` is tedious. LAPIS provides an automatic shortcut. When a Tcl command typed into the Command bar returns a nonempty result, regardless of whether it used `doc`, the result becomes the current document and is displayed in the browser pane. Thus, entering the command

```
expr 5*5+2
```

would display `27` in the browser pane. When an interactive command returns the empty string, however, LAPIS does not change the current document, but instead pops up a dialog box indicating that “The command returned no data.” LAPIS treats empty results differently from nonempty results because many Tcl commands always return empty strings, and it seems unproductive to clear the browser pane when one of these commands is issued. This behavior also matches the behavior of many web browsers, which display a message like “The server returned no data” rather than an empty page. The user can override this behavior by explicitly passing the result of the Tcl command to `doc`, which always sets the current document even if the result is empty.

8.13 Document Metadata

A LAPIS document is essentially a string of content augmented with some metadata represented by name-value pairs. The current document’s properties may be accessed by the `property` command. For example,

```
property url
```

returns the current document’s `url` property. The command

```
property -set base http://www.cs.cmu.edu/
```

sets the current document’s `base` property. The command

```
property -list
```

[5]In fact, this property is one of the appealing features of Tcl, since it allows users to define new control abstractions in the same way that they would define new procedural abstractions.

returns a Tcl list of all the property names defined on the current document.

LAPIS defines several standard properties:

- `content-type`: the document's MIME type. Typical values are `text/plain` for plain text and `text/html` for HTML.
- `command`: the command that created the document.
- `url`: the URL from which the document was loaded.
- `base`: the URL to which relative links refer, initialized from the HTML `<base>` tag.
- `process`: the `java.lang.Process` object that represents the process that generated the document.
- `stdout`: the standard-output document, if the document was created by a process.
- `stderr`: the standard-error document, if the document was created by a process.

Metadata properties are used in many ways in the LAPIS user interface. When a document is first displayed in the browser pane, its `content-type` property helps determine whether it should be shown as rendered HTML or plain text. When a relative URL appears in a hyperlink or inlined image (such as `<img src=/images/go.gif>`), the `base` and `url` properties are used to resolve it into an absolute URL. Documents are listed in the browsing history by their `command` property (or `url` property, if the `command` property is not found).

LAPIS commands automatically copy the `base` and `content-type` properties from the input document to the output document. The only exception to this rule is the `extract -as` *`type`* command, which sets the `content-type` of its output document to match *`type`*. If the input document lacks a `base` property, then the input document's `url` property is copied to the output document's `base` property. As a result of these rules, when a LAPIS command is applied to a web page, the output page will always be displayable as a web page with valid hyperlinks and inlined images, even if the URLs are relative.

When a LAPIS document is converted to a string, however, its metadata is lost. There are two ways this can happen: when the document is saved to a file, which only saves its content; and when the document is processed by other Tcl commands, which treat it as a string. The `relocate` command allows the `base` property to survive this conversion. Running

```
relocate
```

takes the current document's `base` property (or `url` property, if no `base` is found) and creates a new document with an explicit `<base>` tag inserted in the document content.

Another solution to this problem would have all LAPIS commands transform the content so that hyperlinks and inlined images are automatically preserved. This could be done either by adding the `<base>` tag automatically (essentially running `relocate` automatically after every LAPIS command), or by converting all relative URLs into absolute URLs. This solution would obviate the need for the `base` metadata property. This approach seems too draconian, however. It effectively hardcodes an absolute location for the web page, which is undesirable if the user is editing a web page that might be visited through different URLs (e.g., the user might access it by a `file:` URL for editing, but web site visitors would use an `http:` URL). LAPIS adopts the philosophy that the content of documents should be under the user's control, and LAPIS commands should not change document content unless explicitly directed to do so.

The loss of the `content-type` property is not as serious, because it can usually be guessed. For example, when LAPIS loads a document from a filename ending `.htm` or `.html`, it infers that the document has content-type `text/html`. To guess the content type of a string obtained some other way, e.g., from the output of a Tcl command or an external program, LAPIS examines the beginning of the string for either `<html>` or `<!doctype html` (ignoring case and whitespace). If either of these strings is found, then LAPIS guesses content type `text/html`; otherwise, it assumes `text/plain`.

8.14 Command Line Invocation

LAPIS scripts can be invoked by external programs using the LAPIS command line interface. The design of this interface was inspired by the command line interfaces of awk and Perl.

The LAPIS command line interface is a batch interface, which does not display any GUI. It accepts input from command line arguments, files, and/or standard input, and sends output to standard output or files. To use the interface, the caller invokes LAPIS with three kinds of command-line arguments:

- the *invocation mode*, which can be any one of `-pipe`, `-inplace`, `-batch`, or `-argv`. If the mode is omitted, the default mode is `-pipe`. The modes are defined below.
- a script to run, which be specified directly on the command line (`-e` *`script`*) or loaded from a file (`-f` *`scriptfile`*).
- zero or more additional arguments, whose interpretation depends on the invocation mode. For most modes, these arguments are files or URLs; for the `-argv` mode, they are arbitrary string arguments that can be accessed by the LAPIS script.

The invocation modes are explained below.

Pipe Mode

Using one of the following command lines invokes LAPIS in pipe mode:

```
lapis -pipe -e script      file/URL file/URL ...
lapis -pipe -f scriptfile  file/URL file/URL ...
```

In response, the LAPIS commands in *`script`* or loaded from *`scriptfile`* are run on every file or URL listed on the command line, and the resulting documents are printed to standard output. To be precise, for every *`file/URL`*, LAPIS runs the following commands:

```
file/URL     # load the file or URL
script       # run the user's script
puts [doc]   # print result to standard output
```

For example, the following command prints all the comments in a set of Java source files:

```
lapis -pipe -e "extract Comment" *.java
```

Note that pipe mode is the default invocation mode, so `-pipe` can be omitted. This example prints the current price of Lucent stock:

```
lapis -e "extract {Number in Bold in Table starting ’Symbol’}" \
      "http://finance.yahoo.com/q?s=lu&d=v1"
```

If no *`file/URL`* arguments are specified, then pipe mode takes its input from standard input. This allows LAPIS to be used as a filter in a Unix pipeline:

```
ps -aux | lapis -e "sort Process -by CPUShare \
                              -order reverse numeric"
```

Standard input may also be specified explicitly on the command line as `-` (dash).

Pipe mode is analogous to the default modes of awk and Perl, which apply the user’s script to every file on the command line.

Inplace Mode

When LAPIS is invoked one of the following ways:

```
lapis -inplace -e script      file file ...
lapis -inplace -f scriptfile  file file ...
```

then LAPIS runs the script on every file listed on the command line, and saves the resulting document back to the same filename. The effect is the same as this sequence of commands:

```
file       # load the file
script     # run the user’s script
save file  # save back to the same filename
```

The `save` command automatically creates a backup file named *`file`*`~`. At present, inplace mode only works on files, not `http:` or `ftp:` URLs.

Inplace mode is useful for performing a global find-and-replace across a group of files. For example, the following command would replace identifiers named `fetch` with `get` across a set of Java files:

```
lapis -inplace -e "replace {Identifier = ’fetch’} get" *.java
```

Perl also provides in-place processing using its `-i` option.

Batch Mode

Batch mode is similar to pipe mode in that it takes zero or more files or URLs:

```
lapis -batch -e script      file/URL file/URL ...
lapis -batch -f scriptfile  file/URL file/URL ...
```

Unlike pipe mode, however, batch mode is silent by default. It simply loads the file or URL and runs the user's script:

```
file/URL     # load the file or URL
script       # run the user's script
```

Batch mode assumes that the user's script generates its own output or causes some other side-effect. The command may use `puts` to print something to standard output, or `save` to save a document to disk, or may invoke some other command (or a web transaction) that has a side-effect.

Argument Mode

The last command line mode is argument mode, indicated by `-argv`:

```
lapis -argv -e script      arg arg ...
lapis -argv -f scriptfile  arg arg ...
```

Unlike the previous modes, argument mode does not interpret the additional arguments. Instead, the arguments are placed in the Tcl variable `argv`. The user's script is responsible for looking at the `argv` variable to obtain any arguments it needs, do its processing, and then produce its output.

Argument mode is used to invoke self-contained programs written in the LAPIS scripting language. For example, this program searches an online dictionary for the definition of every word passed as an argument:

```
  # look up dictionary definitions
  # for each word given in arguments
foreach word $argv {

     # get dictionary's search page
  http://work.ucsd.edu:5141/cgi-bin/http_webster

     # fill in and submit the search form
  enter Textbox $word
  submit

     # extract the word's definition
  extract {
    from Heading starting "From"
      to point just before Rule
  }
```

```
       # strip out HTML tags
    omit Tag

       # print definition to output
    puts [doc]
}
```

If this program were stored in a file named `lookup`, then it could be invoked by

```
lapis -argv -f lookup word word ...
```

For easier invocation, Unix allows scripts like `lookup` to begin with a line of the form `#!` that specifies the script interpreter to run. For LAPIS, this line looks like:

```
#!/home/rcm/lapis/bin/lapis-script -argv
```

where `/home/rcm` should be replaced by the location of the LAPIS installation. Using this trick, the `lookup` script can be run from the Unix command prompt, just like any other Unix script:

```
% lookup zygote
From WordNet (r) 1.6 (wn)
zygote
    n : the cell resulting from the union of an ovum and a
        spermatozoon (including the organism that develops
        from that cell) [syn: {fertilized cell}]
```

Chapter 9

Selection Inference

Chapter 7 described three ways to make selections in LAPIS — using the mouse, choosing a named pattern from the library pane, and writing a TC pattern. This chapter[1] describes a fourth way — inferring selections from examples.

Two techniques are described:

- *Selection guessing* is the most general technique. It takes positive and negative examples from the user and infers a TC pattern consistent with the examples, which is then used to make the selection. At any time, the user can invoke an editing operation or a menu command on the current selection, start a fresh selection somewhere else, or tell the system to stop making inferences and add or remove selections manually with the mouse.

- *Simultaneous editing* is a form of selection guessing specialized for a common case in repetitive text editing: applying a sequence of edits to a group of text regions. Simultaneous editing is a two-step process. The user first selects a group of *records*, such as lines or paragraphs or postal addresses. This record selection can be made like any other selection — using the mouse, writing a TC pattern, or using selection guessing. Once the desired records have been selected, the system enters a mode in which inference is constrained to produce exactly one selection in every record. The constraints of simultaneous editing permit fast inference with few examples, so few in fact that simultaneous editing approaches the ideal of one-example inference.

This chapter is divided into three parts. The first section describes the user interface techniques of selection guessing and simultaneous editing. The second section details the implementation of these techniques, showing in particular how region set data structures (Chapter 4) enable substantial preprocessing and fast search for hypotheses. The last section describes two user studies, one of simultaneous editing and the other of selection guessing.

9.1 User Interface

To use selection inference, the user must switch into a different *selection mode*, which affects how mouse selections are interpreted. LAPIS has three selection modes:

[1]Portions of this chapter are adapted from earlier papers [MM01a, MM02].

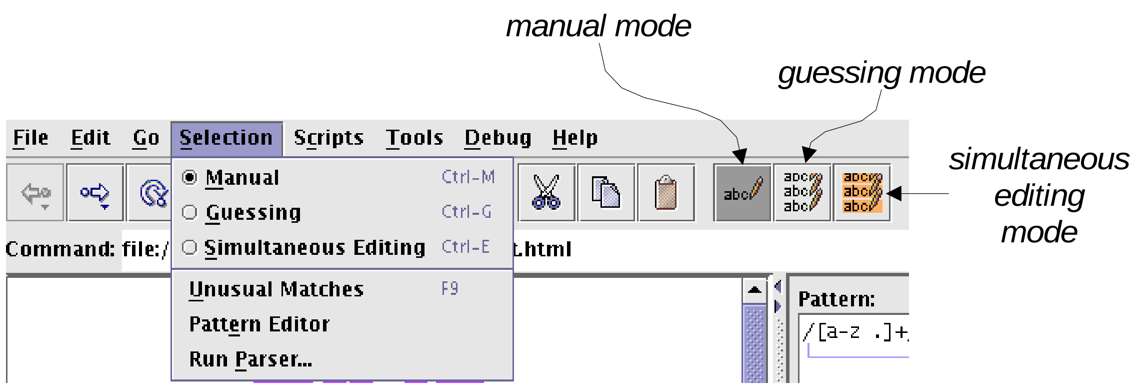


Figure 9.1: The selection mode can be changed with either the Selection menu or toolbar buttons.

- *Manual mode* is the default selection mode. In this mode, mouse selections add and remove regions from the current selection, and no inference is done. Manual selection mode was described in Section 7.4.1.

- *Guessing mode* is the simplest but least efficient inference mode. In this mode, the user's mouse selections are interpreted as positive and negative examples. Whenever the user provides a new example, the system infers a pattern and changes the selection to match its hypothesis.

- *Simultaneous editing mode* is a two-part mode. The first part behaves like guessing mode while the user selects the records. In the second part, the user's selections within the records are treated as positive examples for inference constrained to make exactly one selection per record.

The current selection mode can be changed either by the Selection menu or by a group of toolbar buttons. Both are shown in Figure 9.1.

The selection mode only affects how mouse selections are handled. All other features of LAPIS — editing commands, script commands, menu commands, pattern matching, etc. — can be invoked regardless of the current selection mode.

Introducing modes also introduces the danger of *mode errors*, i.e., performing an operation in the wrong mode. LAPIS tries to alleviate this problem somewhat by making the current selection mode prominent, displaying a dialog pane on the right side of the window (e.g., in Figure 9.2). In simultaneous editing mode, the record set is highlighted in yellow, as a further visual cue to the current mode. These techniques have not been sufficient to eliminate all mode errors, however. Users in the user study occasionally tried to give examples without switching into inference mode. In my own use of LAPIS, I occasionally forget to switch out of inference mode before trying to make a single selection. Selection feedback makes mode errors quickly apparent, however.

One solution to mode errors would be a *spring-loaded* mode, such as a modifier key. For example, holding down the Alt key while making a mouse selection might indicate that the selected region should be used as an example, triggering inference on that selection. Modifier keys offer less visible affordances than a toolbar mode, however, and manual mouse selection already uses several modifier keys (Control and Shift) that might easily be confused with the inference modifier key. Evaluating these tradeoffs on users is left for future work.

9.1.1 Selection Guessing Mode

When the user enters selection guessing mode, a dialog pane appears on the right side of the window (Figure 9.2(a)). The pane contains a help message and a small set of controls. Originally, this dialog was popped up as a modeless dialog box window which floated over the LAPIS window, but the floating window had two problems. First, users found that the window obscured their work, and often felt the need to move or resize it to see what was underneath. Second, some users assumed that the dialog box was modal, and hence had to be dismissed before they could interact with the main LAPIS window again. Both problems were solved by moving the dialog into the LAPIS window, at the cost of obscuring the library pane. The library pane is not usually needed during inference. For the occasional cases when it is, a future version of LAPIS may merely shrink it instead of hiding it completely, using a technique like Mozilla's Sidebar.

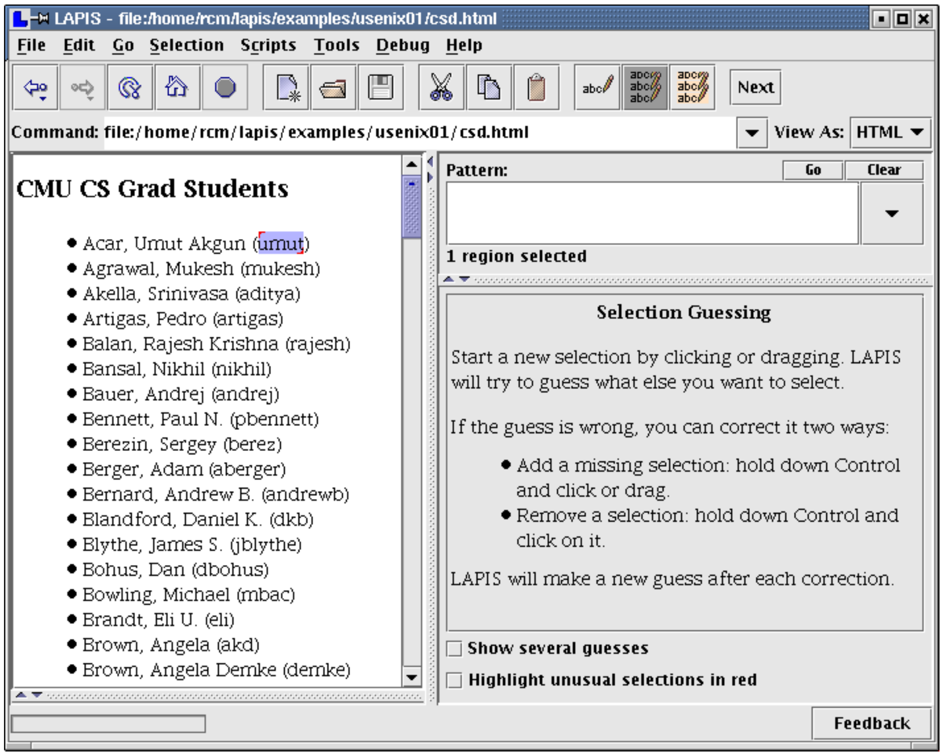
In selection guessing mode, when the user starts a new selection by clicking or dragging, the system uses the selected region as a positive example of the desired selection and infers a TC pattern that is consistent with it. The inferred pattern is displayed both as additional selections in the browser pane, and as a pattern in the pattern pane. Figure 9.2 illustrates this process.

If the inferred selection is wrong, the user can correct it in two ways. The first way is by giving additional examples. Additional positive examples are given by adding regions to the selection — holding down Control while clicking or dragging. Negative examples are given by removing regions from the selection — holding down Control and clicking on the selected region. After each example, the system updates its hypothesis to account for all the positive and negative examples given so far.

In the first design of selection guessing, positive and negative examples were highlighted in various colors, to help the user keep track of which examples had been given. Positive examples were colored dark blue, in order to stand out against the light blue inferred selections, and negative examples were pink. When example highlighting was tested in the user study, however, most users didn't understand what the different colors of blue and pink meant, so the feedback was largely useless. In retrospect, example highlighting seems largely unnecessary. Negative examples are irrelevant once they've been removed from the selection, and it seems unnecessary to recall precisely which selections were positive examples. In any case, users in the study generally gave only a few examples to correct a selection before giving up and trying to make the selection another way. Little can be gained from highlighting a few examples in a special way, aside from visual confusion. Positive and negative example highlighting was subsequently removed from LAPIS.

The second way to correct a selection is to choose an alternative hypothesis. When "Show several guesses" is checked in the dialog pane, the help message is replaced by a list of hypotheses that are consistent with the user's examples (Figure 9.3). Each hypothesis is described by a TC pattern, along with the number of regions it would select and a ranking score, which is described in more detail later in this chapter. By default, the system chooses the highest-ranked hypothesis as its guess. The user can click on any hypothesis in the list to switch to it and see its selection in the browser pane. The hypothesis list does not include *all* possible hypotheses consistent with the user's examples, however. Additional examples may be needed to constrain the hypothesis space sufficiently.

The hypothesis list also includes a "manual selection" choice, which inhibits inference and treats the user's last example as a manual correction on the previous selection. This feature was motivated by user study observations. In principle, if the desired selection lies in the space of


LAPIS - file:/home/rcm/lapis/examples/usenix01/csd.html
File Edit Go Selection Scripts Tools Debug Help
Next
Go
Clear
Feedback

Command: file:/home/rcm/lapis/examples/usenix01/csd.html

View As: HTML

CMU CS Grad Students

- Acar, Umut Akgun (umut)
- Agrawal, Mukesh (mukesh)
- Akella, Srinivasa (aditya)
- Artigas, Pedro (artigas)
- Balan, Rajesh Krishna (rajesh)
- Bansal, Nikhil (nikhil)
- Bauer, Andrej (andrej)
- Bennett, Paul N. (pbennett)
- Berezin, Sergey (berez)
- Berger, Adam (aberger)
- Bernard, Andrew B. (andrewb)
- Blandford, Daniel K. (dkb)
- Blythe, James S. (jblythe)
- Bohus, Dan (dbohus)
- Bowling, Michael (mbac)
- Brandt, Eli U. (eli)
- Brown, Angela (akd)
- Brown, Angela Demke (demke)

Pattern:

1 region selected

Selection Guessing

Start a new selection by clicking or dragging. LAPIS will try to guess what else you want to select.

If the guess is wrong, you can correct it two ways:

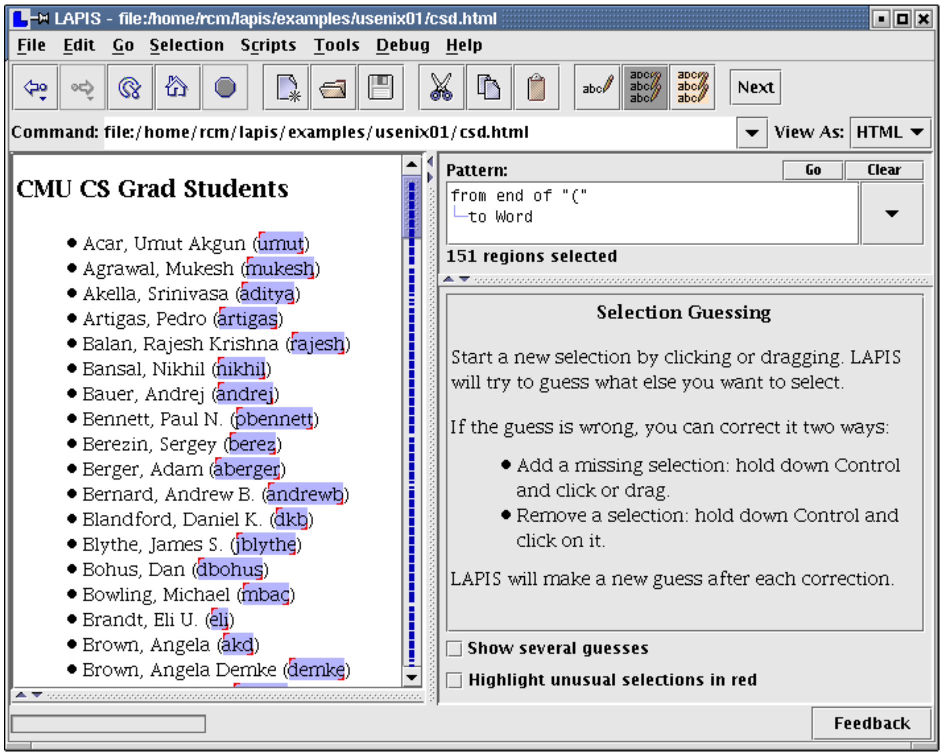
- Add a missing selection: hold down Control and click or drag.
- Remove a selection: hold down Control and click on it.

LAPIS will make a new guess after each correction.

☐ Show several guesses

☐ Highlight unusual selections in red

(a) user selects an example, "umut", with the mouse


LAPIS - file:/home/rcm/lapis/examples/usenix01/csd.html
File Edit Go Selection Scripts Tools Debug Help
Next
Go
Clear
Feedback

Command: file:/home/rcm/lapis/examples/usenix01/csd.html

View As: HTML

CMU CS Grad Students

- Acar, Umut Akgun (umut)
- Agrawal, Mukesh (mukesh)
- Akella, Srinivasa (aditya)
- Artigas, Pedro (artigas)
- Balan, Rajesh Krishna (rajesh)
- Bansal, Nikhil (nikhil)
- Bauer, Andrej (andrej)
- Bennett, Paul N. (pbennett)
- Berezin, Sergey (berez)
- Berger, Adam (aberger)
- Bernard, Andrew B. (andrewb)
- Blandford, Daniel K. (dkb)
- Blythe, James S. (jblythe)
- Bohus, Dan (dbohus)
- Bowling, Michael (mbac)
- Brandt, Eli U. (eli)
- Brown, Angela (akd)
- Brown, Angela Demke (demke)

Pattern:

```
from end of "("
└to Word
```

151 regions selected

Selection Guessing

Start a new selection by clicking or dragging. LAPIS will try to guess what else you want to select.

If the guess is wrong, you can correct it two ways:

- Add a missing selection: hold down Control and click or drag.
- Remove a selection: hold down Control and click on it.

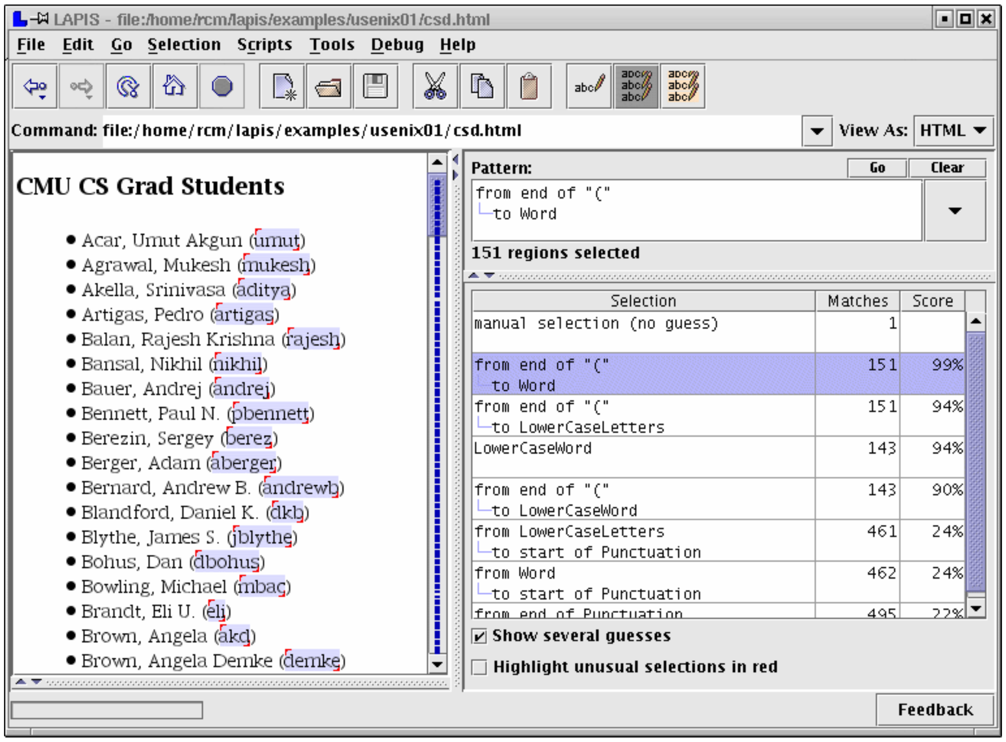
LAPIS will make a new guess after each correction.

☐ Show several guesses

☐ Highlight unusual selections in red

(b) system infers a pattern and selects all matches to it

Figure 9.2: Making a selection by example in selection guessing mode.



LAPIS - file:/home/rcm/lapis/examples/usenix01/csd.html

Command: file:/home/rcm/lapis/examples/usenix01/csd.html

CMU CS Grad Students

- Acar, Umut Akgun (umut)
- Agrawal, Mukesh (mukesh)
- Akella, Srinivasa (aditya)
- Artigas, Pedro (artigas)
- Balan, Rajesh Krishna (rajesh)
- Bansal, Nikhil (nikhil)
- Bauer, Andrej (andrej)
- Bennett, Paul N. (pbennett)
- Berezin, Sergey (berez)
- Berger, Adam (aberger)
- Bernard, Andrew B. (andrewb)
- Blandford, Daniel K. (dkb)
- Blythe, James S. (jblythe)
- Bohus, Dan (dbohus)
- Bowling, Michael (mbac)
- Brandt, Eli U. (eli)
- Brown, Angela (akd)
- Brown, Angela Demke (demke)

```
from end of "("
└to Word
```

151 regions selected

Selection	Matches	Score
manual selection (no guess)	1	
from end of "(" └to Word	151	99%
from end of "(" └to LowerCaseLetters	151	94%
LowerCaseWord	143	94%
from end of "(" └to LowerCaseWord	143	90%
from LowerCaseLetters └to start of Punctuation	461	24%
from Word └to start of Punctuation	462	24%
from end of Punctuation	495	22%

Figure 9.3: Selection guessing mode also offers a list of alternative hypotheses.

learnable hypotheses, then manual selection should be unnecessary, since the user's corrections would eventually converge on the right answer. In practice, however, users have no way to predict how many examples might be needed to learn the desired selection, or whether the system can learn it at all. As a result, as soon as an almost-correct hypothesis appeared, users expressed a desire to make the system stop guessing and let them fix the exceptions manually.

If the desired selection is outside the system's hypothesis space, inference will eventually fail to find a hypothesis consistent with the examples. When inference fails, the system stops updating its hypothesis and displays "unknown selection" as the pattern. The user can continue adding and removing regions from the selection manually. If inference failed because an incorrect example was given, the user can retract the example — e.g., if an incorrect positive example was given, the user gives it as a negative example — and the system will resume trying to make inferences.

Once the desired selection has been made, the user can edit or apply text-processing tools to it, as described in previous chapters. While the user is typing or deleting characters using an inferred selection, no inference is done. When the user starts a new selection with the mouse, the set of examples is cleared and the system generates a fresh hypothesis.

Inferred patterns can be edited by the user and run again. Once an inferred pattern has been edited, however, the system stops doing inference and throws away the user's examples, since the edited pattern may no longer lie in the hypothesis space.

The selection guessing dialog pane includes another checkbox, "Highlight unusual selections in red." When this box is checked, the outliers of the inferred selection are highlighted in red. Outlier highlighting is discussed in detail in Chapter 10.

9.1.2 Simultaneous Editing Mode

Many repetitive editing tasks have a common form: a group of things all need to be changed in the same way. Some examples from the PBD literature include:

- add "[author year]" to bibliographic citations [Mau94]
- reformat baseball scores [Nix85]
- change the styles of all section headings [Mye93]

Each of these tasks can be represented as an iteration over a list of text regions, called *records* for lack of a better name, where the body of the loop performs a fixed sequence of edits on each record. LAPIS addresses this class of tasks with simultaneous editing mode.

The first step in simultaneous editing is describing the record set, which is done by selecting all the records. When the user enters simultaneous editing mode, LAPIS displays a dialog pane (Figure 9.4) which prompts the user to select the records. The user may use the mouse to give positive and negative examples of records, using the same interaction techniques as selection guessing. The user may also enter a pattern to select the records. An experienced user can shortcut this dialog by selecting the records before entering simultaneous editing mode. If at least two nonzero-length regions are selected when the user clicks on the Simultaneous Editing menu command or toolbar button, then this selection is used as the record set.

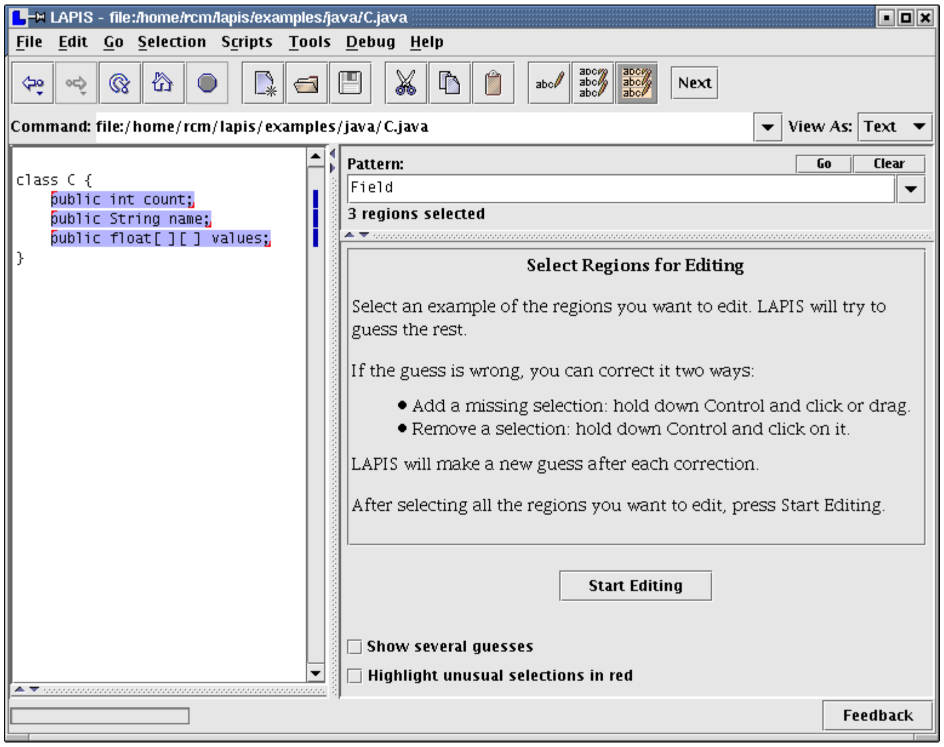
Once the desired record set is selected, the user clicks on the Start Editing button in the dialog pane to enter the second step of simultaneous editing mode. The selected record set becomes highlighted in yellow, and the dialog pane changes to describe the new mode (Figure 9.5). Now, when the user makes a selection in one record, the system automatically infers exactly one selection in every other record. If the inference is incorrect on some record, the user can correct it by holding down the Control key and making the correct selection, after which the system generates a new hypothesis consistent with the new example. As in selection guessing, the user can edit with the multiple selection at any time. The user is also free to make selections outside the yellow records, but no inferences are made from those selections.

Figures 9.6–9.11 illustrate how simultaneous editing can be used to perform a common task in Java and C++ programming: replacing each public field of a class (member variable in C++ terminology) with a private field and a pair of accessor methods `getX` and `setX` that respectively get and set the value of the field.

In Figure 9.6, the user has selected "public" in one record, which causes the system to infer a selection of "public" in the other records as well. The pattern pane displays the TC pattern that matches this inference, `"public"`.

The user deletes this selection, then places the insertion point at the end of the records to start typing the `get` method: first pressing Enter to insert a new line, then indenting a few spaces, then typing "public" to start the method declaration. Following multiple-selection editing rules, the typed characters appear in every record (Figure 9.7). If typos are made, the user can back up and correct them, using all familiar editing operations, including Undo. No inference occurs while the user is typing.

Now the user is ready to enter the return type of the `get` method. The type is different for each variable, so it can't simply be entered at the keyboard. Instead, copy-and-paste must be used.


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
Command: file:/home/rcm/lapis/examples/java/C.java
View As: Text
Pattern:
Go
Clear
Field
Start Editing
Feedback

```
class C {
    public int count;
    public String name;
    public float[ ][ ] values;
}
```

3 regions selected

Select Regions for Editing

Select an example of the regions you want to edit. LAPIS will try to guess the rest.

If the guess is wrong, you can correct it two ways:

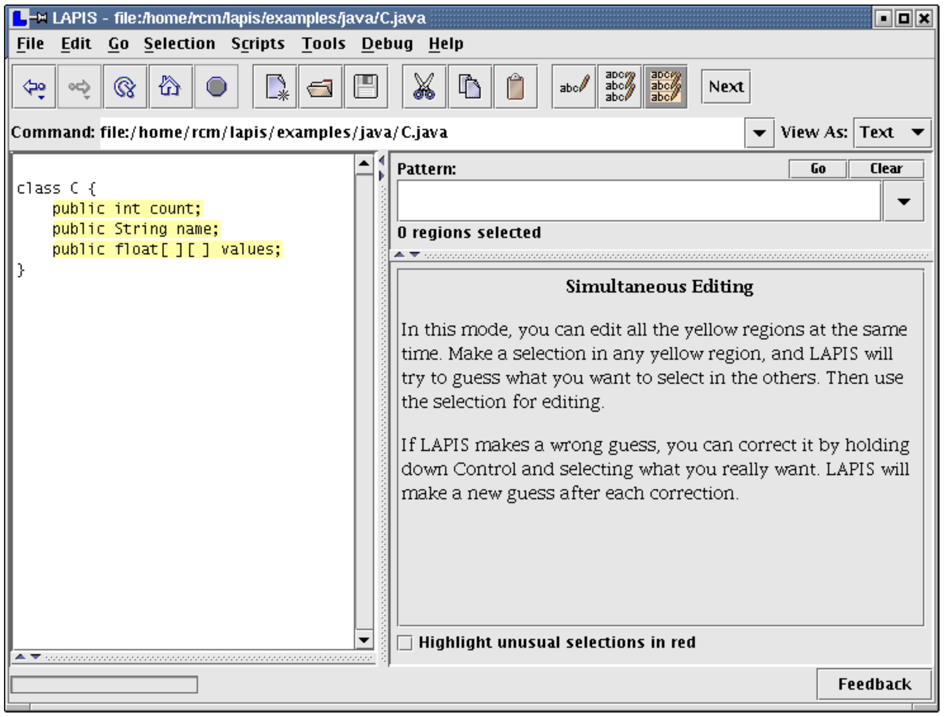
- Add a missing selection: hold down Control and click or drag.
- Remove a selection: hold down Control and click on it.

LAPIS will make a new guess after each correction.

After selecting all the regions you want to edit, press Start Editing.

☐ Show several guesses
☐ Highlight unusual selections in red

Figure 9.4: First step of simultaneous editing: selecting the records using unconstrained inference (selection guessing).


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
Command: file:/home/rcm/lapis/examples/java/C.java
View As: Text
Pattern:
Go
Clear
Feedback

```
class C {
    public int count;
    public String name;
    public float[ ][ ] values;
}
```

0 regions selected

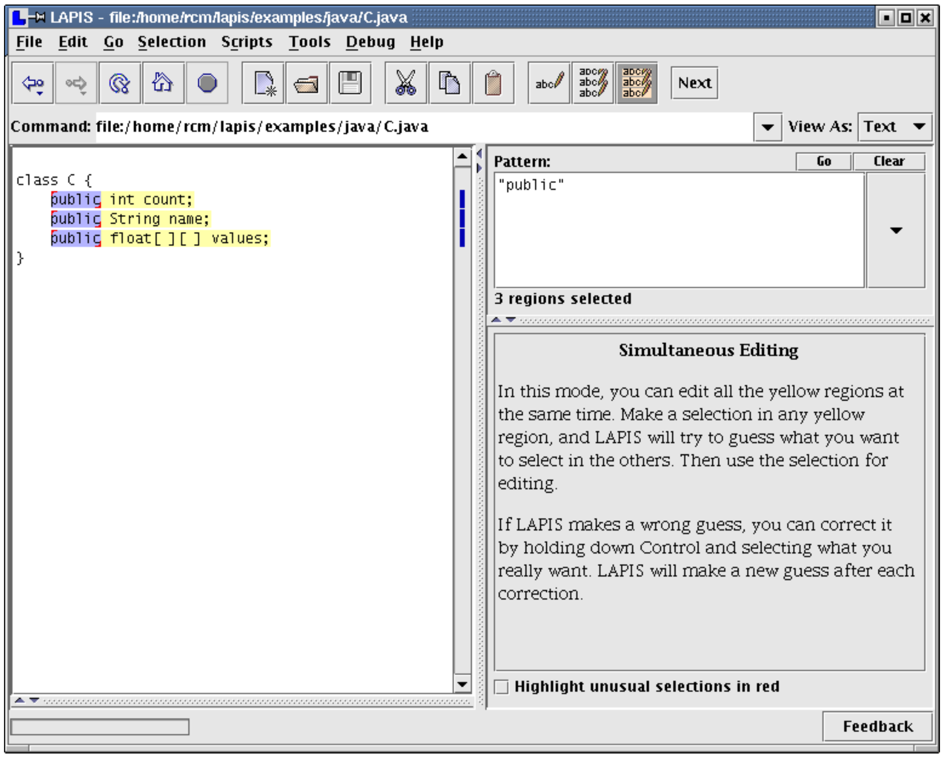
Simultaneous Editing

In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

☐ Highlight unusual selections in red

Figure 9.5: Second step of simultaneous editing: the records turn yellow, and inference is now constrained to make one selection in each record.


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
View As: Text
Pattern:
Go
Clear
Feedback

Command: file:/home/rcm/lapis/examples/java/C.java

```
class C {
    public int count;
    public String name;
    public float[ ][ ] values;
}
```

Pattern: "public"

3 regions selected

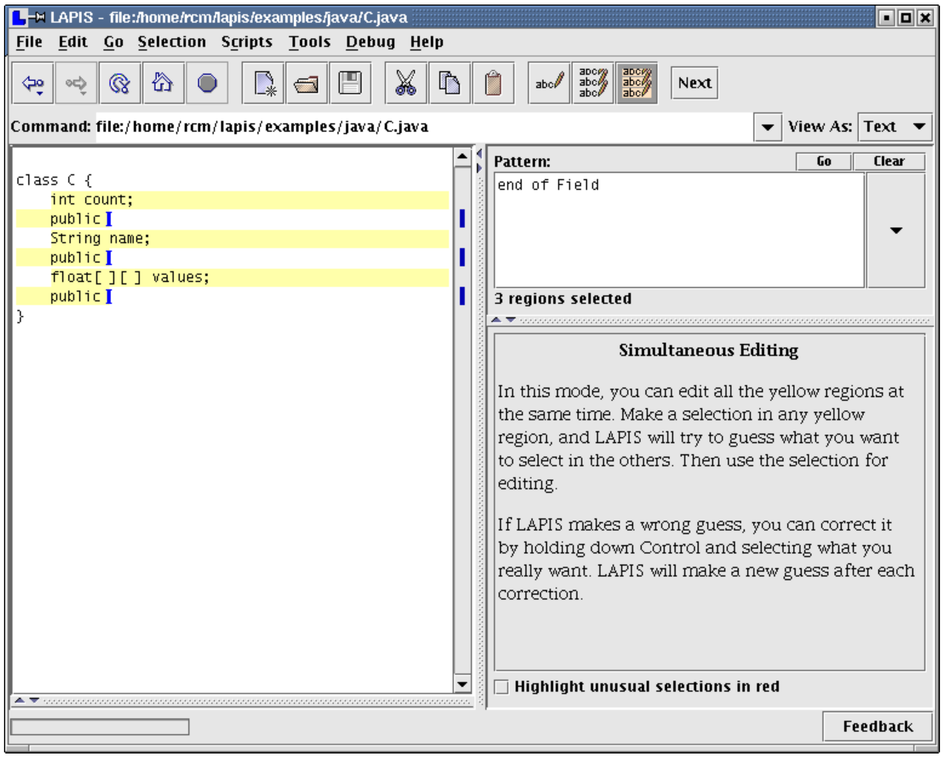
Simultaneous Editing

In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

☐ Highlight unusual selections in red

Figure 9.6: Selecting "public" in one record causes the system to infer equivalent selections in the other records.


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
View As: Text
Pattern:
Go
Clear
Feedback

Command: file:/home/rcm/lapis/examples/java/C.java

```
class C {
    int count;
    public 
    String name;
    public 
    float[ ][ ] values;
    public 
}
```

Pattern: end of Field

3 regions selected

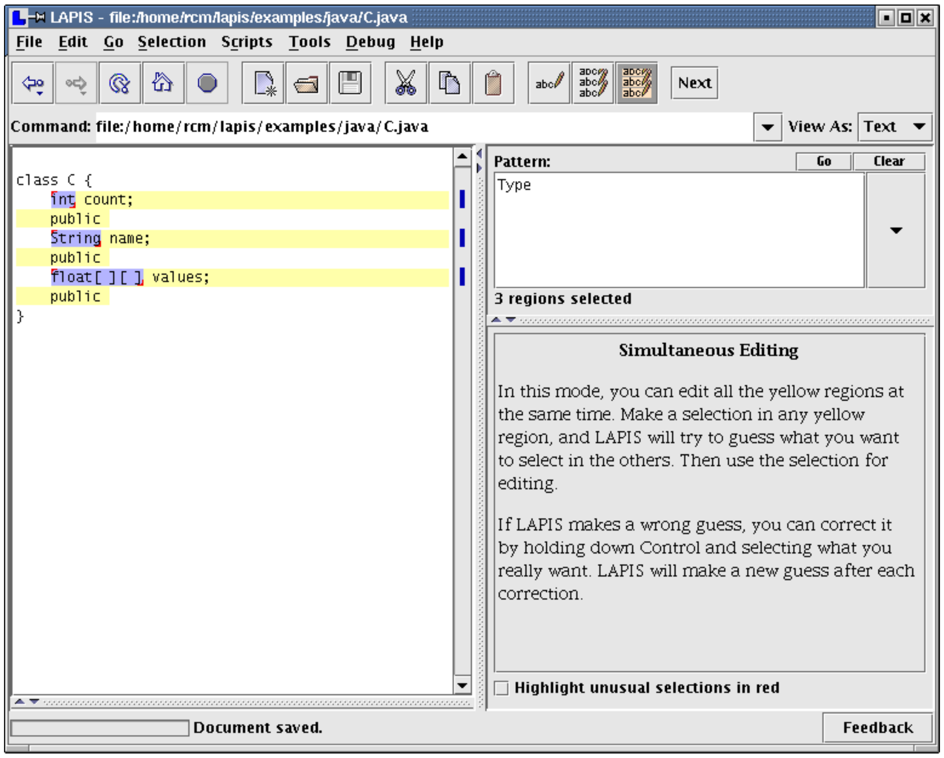
Simultaneous Editing

In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

☐ Highlight unusual selections in red

Figure 9.7: Typing and deleting characters affects all regions.


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
Command: file:/home/rcm/lapis/examples/java/C.java
View As: Text
Pattern:
Go
Clear
Feedback

```
class C {
    int count;
    public
    String name;
    public
    float[ ][ ] values;
    public
}
```

Pattern: Type

3 regions selected

Simultaneous Editing

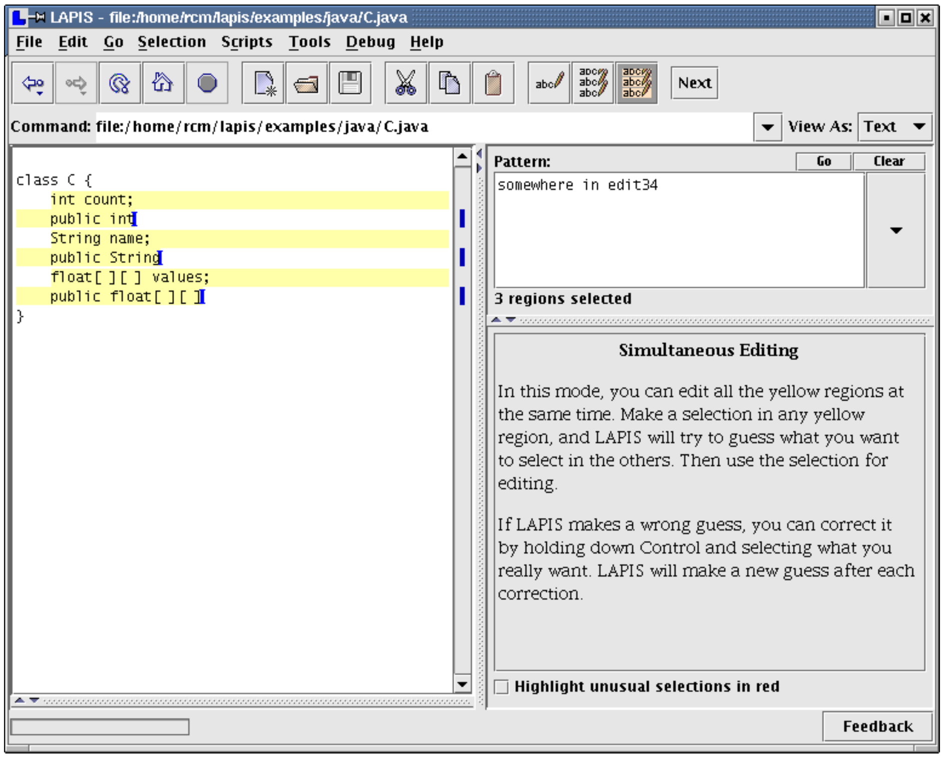
In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

Highlight unusual selections in red

Document saved.

Figure 9.8: The user selects the types and copies them to the clipboard...


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
Command: file:/home/rcm/lapis/examples/java/C.java
View As: Text
Pattern:
Go
Clear
Feedback

```
class C {
    int count;
    public int
    String name;
    public String
    float[ ][ ] values;
    public float[ ][ ]
}
```

Pattern: somewhere in edit34

3 regions selected

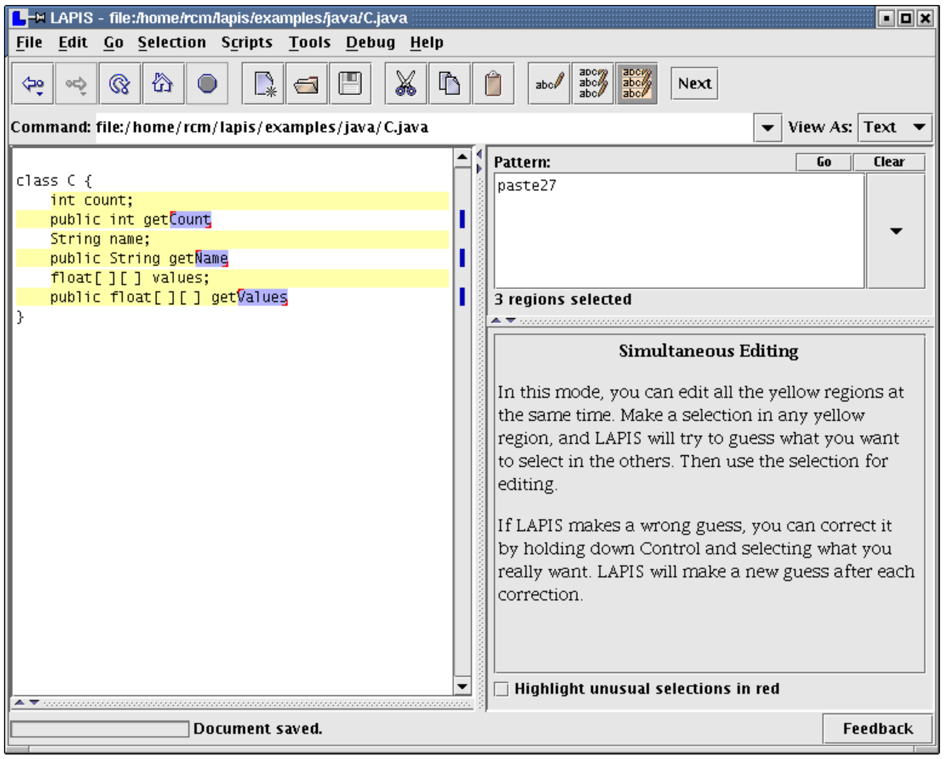
Simultaneous Editing

In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

Highlight unusual selections in red

Figure 9.9: ... and then pastes the copied selections back.


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
Command: file:/home/rcm/lapis/examples/java/C.java
View As: Text
Pattern:
Go
Clear
Feedback

```
class C {
    int count;
    public int getCount
    String name;
    public String getName
    float[ ][ ] values;
    public float[ ][ ] getValues
}
```

paste27

3 regions selected

Simultaneous Editing

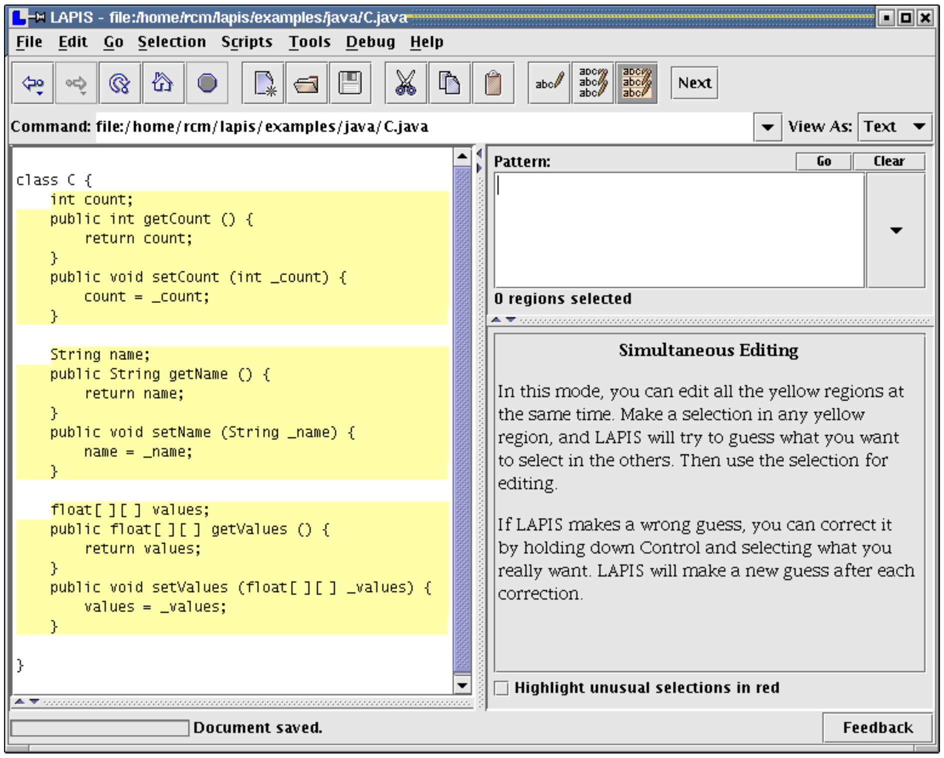
In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

Highlight unusual selections in red

Document saved.

Figure 9.10: Changing the capitalization of the method name.


LAPIS - file:/home/rcm/lapis/examples/java/C.java
File Edit Go Selection Scripts Tools Debug Help
Next
Command: file:/home/rcm/lapis/examples/java/C.java
View As: Text
Pattern:
Go
Clear
Feedback

```
class C {
    int count;
    public int getCount () {
        return count;
    }
    public void setCount (int _count) {
        count = _count;
    }

    String name;
    public String getName () {
        return name;
    }
    public void setName (String _name) {
        name = _name;
    }

    float[ ][ ] values;
    public float[ ][ ] getValues () {
        return values;
    }
    public void setValues (float[ ][ ] _values) {
        values = _values;
    }

}
```

0 regions selected

Simultaneous Editing

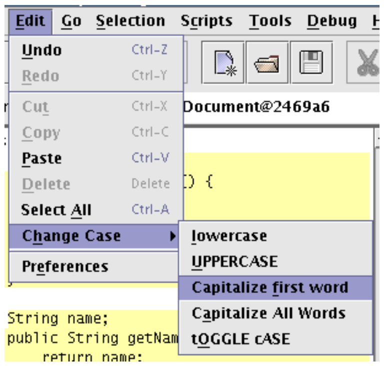
In this mode, you can edit all the yellow regions at the same time. Make a selection in any yellow region, and LAPIS will try to guess what you want to select in the others. Then use the selection for editing.

If LAPIS makes a wrong guess, you can correct it by holding down Control and selecting what you really want. LAPIS will make a new guess after each correction.

Highlight unusual selections in red

Document saved.

Figure 9.11: The final outcome of simultaneous editing.



```
String name;
public String getNam
    return name;
```

Figure 9.12: Commands on the Change Case menu change the capitalization of the selection.

The user first selects the type of one of the records, such as the “int” in “int x”. The system infers the pattern `Type` and selects the types in the other fields (Figure 9.8). This shows an important difference between simultaneous editing and selection guessing. In selection guessing, many other hypotheses would also be consistent with the user’s example: `"int"`, `2nd Word in Line`, `2nd from last Word in Line`, etc. In simultaneous editing, some of these hypotheses can be discarded immediately because they do not make exactly one selection in each record. For example, the literal string `"int"` does not appear in every record. Other generalizations are less preferable because they are more complicated than `Type`.

The user then copies the selection to the clipboard, places the insertion point back after “public”, and pastes the clipboard. Following multiple-selection editing rules, the system copies a list of strings to the clipboard, one for each record, and pastes the appropriate string back to each record (Figure 9.9). The one-selection-per-record bias helps here, too. Since the number of inferred selections always equals the number of records, there is no danger of trying to paste n copied selections back to m insertion points (which would cause LAPIS to pop up an error dialog, as explained in Section 7.5).

Note that the inference in Figure 9.9 is described as `somewhere in edit34`, which is not a valid TC pattern. This is another difference between selection guessing and simultaneous editing. In selection guessing, all inferences are TC patterns. In simultaneous editing, however, when the user makes a selection in new text — text that has been typed or pasted since entering simultaneous editing mode — LAPIS does not bother to search for a TC pattern describing the selection. Instead, it simply infers the selection that originally created the text, and displays an approximate description of the selection in the pattern pane. This technique has significant performance advantages — in particular, it eliminates the need to continually reparse the document as the user is editing, since new text need not be parsed. The tradeoff is that some inferences do not produce valid TC patterns, which is not ideal for self-disclosure of the TC pattern language. Fast response time seems to outweigh the value of self-disclosure, at least in this case.

Next, the user copies and pastes the name of the variable to create the method name (Figure 9.10). The lowercase variable name must be capitalized by applying a menu command that capitalizes the current selection (Figure 9.12). Any menu command that applies to a selection can be used in simultaneous editing mode.

The rest of the `get` and `set` methods are defined by more typing and copy-and-paste commands, until the final result is achieved (Figure 9.11). The user then switches back to manual selection mode, and the yellow record highlighting disappears.

Simultaneous editing is more constrained than selection guessing, because its hypotheses must have exactly one match in every record. But the one-selection-per-record bias delivers some powerful benefits. First, it dramatically reduces the hypothesis search space, so that fewer examples are needed to reach the desired selection. Second, the hypothesis search is faster. Since the record set is specified in advance, LAPIS can preprocess it to find common substrings and library patterns that occur at least once, significantly reducing the space of features that can be used in hypotheses. As a result, where selection guessing might take several seconds to deliver a hypothesis, simultaneous editing takes 0.4-0.8 sec, making it more suitable for interactive editing. Finally, the one-selection-per-record constraint makes editing semantically identical to single-selection editing on each record. In particular, a selection copied from one place can always be pasted somewhere else in the record, since the source and target are guaranteed to have the same number of selections.

The keyboard can also be used to make selections in simultaneous editing mode. When the user presses an arrow key, the system first clears most of the current selection, leaving only the first example selected. This selection is then moved by the arrow key in the same way that a conventional single-selection text editor would move it. The resulting selection is then treated as an example to infer a new selection.

The practical result of this behavior is that the user can move the selection with the keyboard while inference keeps the selections synchronized. For example, suppose the user places the cursor before "int" in Figure 9.5, causing the system to infer a selection `start of Type`. Pressing the right arrow key three times moves the initial example to the end of "int", causing the system to infer the selection `end of Type`. A more naive approach to multi-cursor keyboard navigation would move all the cursors three characters to the right in lock step — leaving one in the middle of "String" and the other in the middle of "float".

Most conventional keyboard navigation is supported, including Home, End, Page Up, Page Down, holding down Control to move by whole words, and holding down Shift to select a region. The keyboard cannot currently be used to give multiple examples; doing so would require decoupling the keyboard cursor from the selection, so that it can be moved around independently to add and remove selections.

9.2 Implementation

This section describes the algorithms used to infer selections from examples. Each mode, selection guessing and simultaneous editing, uses a different algorithm. Like most learning algorithms, both algorithms essentially search through a space of hypotheses for a hypothesis consistent with the examples, but each algorithm uses a different hypothesis space and a different method of ranking hypotheses, reflecting the different biases of the two modes.

In addition, the simultaneous editing algorithm does considerably more preprocessing than the selection guessing algorithm. Preprocessing allows much of the hypothesis space to be pruned away before the user even starts giving examples, which allows each selection to be inferred faster. The tradeoff is that the preprocessed information falls out of date as soon as the user edits the document, so some extra work has to be done to refresh it. Furthermore, because of preprocessing,

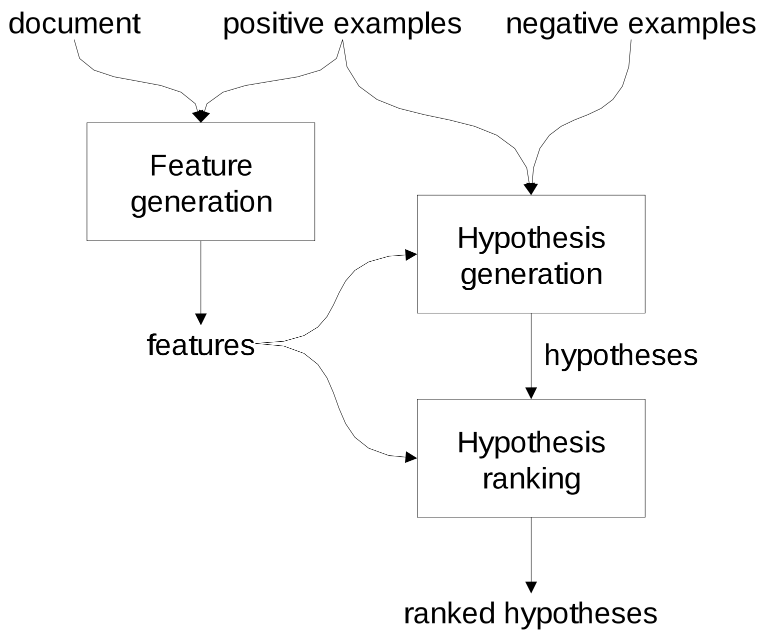


Figure 9.13: Block diagram of selection-guessing algorithm.

simultaneous editing does not always infer a TC pattern; sometimes its inference is simply a region set with no corresponding TC pattern. More will be said about this issue below.

9.2.1 Selection Guessing Algorithm

The inputs to the selection guessing algorithm are a document, a set of positive examples expressed as a region set, and a set of negative examples, also a region set. The output is a ranked list of hypotheses, each of which is a TC pattern that matches the positive examples and excludes the negative examples.

In this algorithm, a *feature* is a TC expression of the form `op F`, where `op` is one of the TC operators `equals`, `just before`, `just after`, `starting`, `ending`, `in`, or `contains`, and `F` is either a literal or a pattern identifier. A *hypothesis* is a Boolean conjunction of features.

The algorithm has three main parts, illustrated in Figure 9.13:

- *Feature generation* takes the document and the positive examples and produces a collection of features that match all the positive examples.

- *Hypothesis generation* takes the features and the positive examples and returns a list of hypotheses that match all the positive examples and exclude the negative examples.

- *Hypothesis ranking* takes the hypotheses and ranks them by a heuristic.

Each part is described below.

Feature Generation

Two kinds of features are generated: *library features* derived from the pattern library, and *literal features* discovered by examining the text of the positive examples.

Library features are generated by searching for every named pattern in the pattern library, to produce the set of regions matching each pattern. This is actually the most time-consuming part of the inference algorithm. Then, the seven relational operators `equals`, `just before`, `just after`, `starting`, `ending`, `in`, and `contains` are applied to each library region set, producing a region set representing all the regions that have the feature. Since these relations are represented by rectangle collections, generating these region sets is fast (Chapter 4). The region set for each feature is then intersected with the region set representing the positive examples. Features which don't match all the positive examples are discarded. Since the algorithm only learns monotone conjunctions of features, only features that match all the positive examples are useful for forming hypotheses.

Literal features are generated by combining the relational operators with literal strings. Instead of the generate-and-test approach used for the library features, however, the generation of literal features is driven by the text of the positive examples. Since useful features must match all the positive examples, the generation algorithm searches for *common substrings*, i.e., substrings that occur in or around all the positive examples. The generation technique for each relational operator is slightly different. In the description that follows, the ith positive example is denoted by the region $x_i[y_i]z_i$, where the whole document is the string $x_iy_iz_i$ and y_i is the part of the document selected by the region.

- `equals`: If all y_i are identical, i.e. $y = y_i$ for all i, then generate the feature `equals "`*`y`*`"`.
- `starting`: Find the common prefixes of all the y_i, i.e., all strings p such that p is a prefix of every y_i. If two prefixes p and pq are equivalent in the sense that every occurrence of p in the document is followed by q, then keep only the shorter prefix p. For example, "http" is equivalent to "http://" if every occurrence of "http" in the document is followed by "://". For all remaining prefixes p, generate the feature `starting "`*`p`*`"`.
- `ending`: Generate the common suffixes of the y_i, using an algorithm analogous to `starting`.
- `just before`: Generate the common prefixes of the z_i.
- `just after`: Generate the common suffixes of the x_i.
- `contains`: Find all common substrings of the y_i using a *suffix tree* [Gus97]. A suffix tree is a path-compressed trie into which all suffixes of a string have been inserted. In this case, the suffix tree represents the set of common substrings of all y_i examined so far. Figure 9.14 illustrates the algorithm. The algorithm starts by building a suffix tree from y_1. (y_1 may be any positive example, but it's useful to let y_1 be the shortest y_i to minimize the size of the initial suffix tree.) For each remaining y_i ($2 \leq i \leq n$), the suffixes of y_i are matched against the suffix tree one by one. Each tree node keeps a count of the number of times it was visited during the processing of y_i. This count represents the number of occurrences in y_i of the substring represented by the node. After processing y_i, all unvisited nodes are pruned from

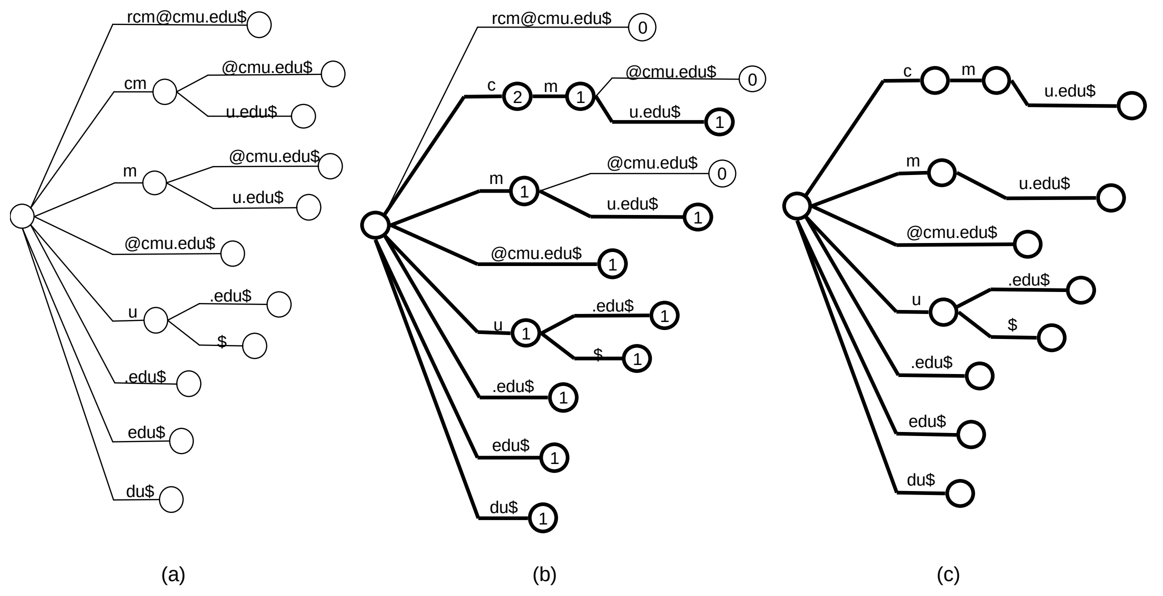


Figure 9.14: Finding common substrings using a suffix tree. (a) Suffix tree constructed from first example, `rcm@cmu.edu`; $ represents a unique end-of-string character. (b) Suffix tree after matching against second example, `ljc@cmu.edu`. Each node is labeled by its visit count. (c) Suffix tree after pruning nodes which are not found in `ljc@cmu.edu`. The remaining nodes represent the common substrings of the two examples.

the tree, since the corresponding substrings never occurred in y_i. After processing all y_i in this fashion, the only substrings left in the suffix tree must be common to all the y_i. Generate a feature `contains "s"` for each such substring s.

- `in`: No literal features of this form are generated, because `in "literal"` is usually redundant with an `equals` feature.

Hypothesis Generation

After generating features that match the positive examples, the algorithm forms conjunctions of features to produce hypotheses consistent with all the examples. Since a selection must have a clearly defined start point and end point, not all conjunctions of features are useful hypotheses, so the algorithm reduces the search space by forming *kernel hypotheses.* A kernel hypothesis takes one of the following forms:

- a single feature which fixes both the start and end, i.e. `equals F`. As a kernel hypothesis, this feature is represented simply by the pattern `F`.

- a conjunction of a start-point feature (`starting F` or `just after F`) with an end-point feature (`ending G` or `just before G`). As a kernel hypothesis, this conjunction is represented as a TC pattern of the form

```
        from start/end of F
          to start/end of G
```

All possible kernel hypotheses are generated from the feature set. Kernel hypotheses that are inconsistent with the positive examples are then discarded.

If there are negative examples, then additional features are added to each kernel hypothesis to exclude them. Features are chosen greedily to exclude as many negative examples as possible. For example, after excluding negative examples, a kernel hypothesis `Link` might become the final hypothesis

```
    Link contains "cmu.edu"
         just-before Linebreak
```

Kernel hypotheses that cannot be specialized to exclude all the negative examples are discarded.

This simple algorithm is capable of learning only monotone conjunctions. This is not as great a limitation as it might seem, because many of the concepts in the pattern library incorporate disjunction (e.g. `UppercaseLetters` vs. `Letters` vs. `Alphanumeric`). It is easy to imagine augmenting or replacing this simple learner with a disjunctive normal form learner, such as the one used by Cima [Mau94].

Hypothesis Ranking

After generating a set of hypotheses consistent with all the examples, there remains the problem of choosing the best hypothesis — in other words, defining the *preference bias* of the learner. Most learning algorithms use Occam's Razor, preferring the hypothesis with the smallest description, in this case, the fewest number of features in its conjunction. Since hypotheses can include library features, however, many hypotheses seem equally simple. Which of these hypotheses should be preferred: `Word`, `JavaIdentifier`, or `JavaExpression`? The algorithm supplements Occam's Razor with a heuristic I call *regularity*.

The regularity heuristic was originally designed for inferring record sets for simultaneous editing. It is based on the observation that simultaneous editing records often contain *regular features*, features that occur a fixed number of times in each record. For instance, most postal addresses contain exactly one postal code and exactly three lines. Most HTML links have exactly one start tag, one end tag, and one URL.

It is easy to find features that occur a regular number of times in all the positive examples. Not all of these features may be regular in the entire record set, however, so the algorithm finds a set of *likely regular features* by the following procedure. For each feature `contains F` that is regular in the positive examples (i.e., where F occurs exactly n_F times in each positive example), count the number of times D_F that F occurs in the entire document. Assuming for the moment that F is a regular feature that occurs *only* in records, then there must be D_F/n_F records in the entire document. Call D_F/n_F the *record count prediction* made by F.

Let M be the record count predicted by the *most* features, in other words, the mode of the record count predictions for all features F. If M is unique and integral, then the *likely regular features* are the features that predicted M. If M is not unique, or M is not an integer, then the algorithm abandons the regularity heuristic, and falls back to Occam's Razor, ranking hypotheses by the number of features.

The upshot of this procedure is that a feature is kept as a likely regular feature only if other features predict exactly the same number of records. Features which are nonregular, occurring fewer times or more times in some records, will usually predict a fractional number of records and be excluded from the set of likely regular features.

For example, suppose the user is trying to select the peoples' names and userids in the list below, and has given the first two items as examples (shown underlined):

<u>Acar, Umut (umut)</u>
<u>Agrawal, Mukesh (mukesh)</u>
Balan, Rajesh Krishna (rajesh)
Bauer, Andrej (andrej)

The two examples have several regular features in common, among them:

- `","` (comma), which occurs exactly once in each example;
- `CapitalizedWord`, which occurs twice;
- `Word`, which occurs 3 times;
- `Parentheses`, which occurs once.

Note that `CapitalizedWord` and `Word` are regular only in the two examples, not in the entire set of names.

Computing the record count prediction N_F/n_F for these features gives:

- comma: 4
- `CapitalizedWord`: 4.5
- `Word`: 4.33
- `Parentheses`: 4

The record count predicted by the most features is 4, so the likely regular features would be comma and `Parentheses`. This example is oversimplified, since the pattern library would find other features as well.

In addition to features of the form `contains` *`F`*, which look for regularity inside the examples, the algorithm also checks for regularity in features describing the context of the examples — before, after, and around.. The features `just before` *`F`* and `just after` *`F`* are considered regular if every positive example contains the same number of occurrences of *`F`* (just like `contains` *`F`*). The feature `in` *`F`* is considered regular if every positive example is in a different instance of *`F`*.

Likely regular features are used to test hypotheses by assigning a higher preference score to a hypothesis if it is in greater agreement with the likely regular features. A useful measure of agreement is the *category utility*, which was also used in Cima [Mau94]. The category utility of a hypothesis H and a feature F is given by

$$\begin{aligned} U(H,F) &= P(H|F)P(F|H) \\ &= \frac{P(H \cap F)^2}{P(H)P(F)} \end{aligned}$$

where $P(F)$ is the probability of having feature F, $P(H)$ is the probability of satisfying the hypothesis H, and $P(H \cap F)$ is the probability of satisfying both the feature and the hypothesis. If a hypothesis and a feature are in perfect agreement, then the category utility is 1; if they are logical complements, then the category utility is 0. The probabilities can be estimated by drawing a sample of size N and counting the number of instances that satisfy H, F, or both:

$$\begin{aligned} U(H,F) &= \frac{(N_{HF}/N)^2}{(N_H/N)(N_F/N)} \\ &= \frac{N_{HF}^2}{N_H N_F} \end{aligned}$$

When category utility is used in selection guessing, these counts are defined as follows:

- N_H is the number of matches to the hypothesis H;
- $N_F = D_F/n_F$ is the number of records predicted by likely regular feature F;
- N_{HF} is the number of matches to hypothesis H that contain exactly n_F occurrences of likely regular feature F.

For each hypothesis, the category utility is averaged across all likely regular features to compute a score between 0 and 1, which is shown (as a percentage) in the Score column in Figure 9.2(b).

9.2.2 Simultaneous Editing Algorithm

We now turn to the algorithm used in simultaneous editing mode. The inputs to the algorithm are a document, a set of records, and a set of positive examples with at most one example per record. Negative examples are not used in this algorithm. The output is a region set that selects exactly one region in every record, plus a human-readable description of the region set, which is usually but not always a TC pattern.

The algorithm is split into three parts (Figure 9.15):

- *Feature generation* takes the set of records as input and generates a list of useful features as output. Unlike the selection guessing algorithm, which repeats feature generation every time an new example is given, feature generation takes place only once, when the user first enters simultaneous editing mode.
- *Hypothesis generation* takes the positive examples and the feature list and searches for a region set consistent with the examples. The search is repeated whenever the user gives a new positive example.

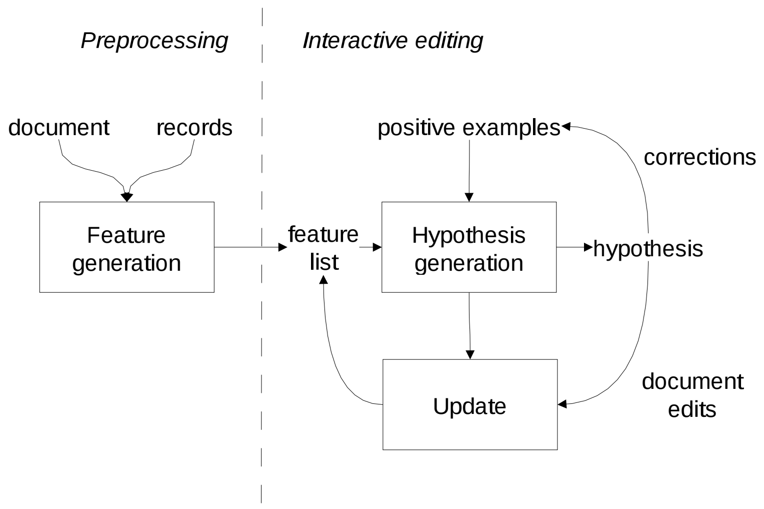


Figure 9.15: Block diagram of simultaneous editing algorithm.

- *Update* occurs when the user edits the records by inserting, deleting, or copying and pasting text. The update algorithm takes the user's edit action as input and modifies the feature list to update it.

Each of these parts is described below.

Feature Generation

Simultaneous editing uses a different set of features than selection guessing. In simultaneous editing, a feature is a region set that has at least one region in each record, and no regions outside any records. Initially, the algorithm generates two kinds of features: *library features* derived from the pattern library, and *literal features* discovered by examining the text of the records.

Library features are found by matching every named pattern in the pattern library, then discarding any patterns that do not have at least one match in every record. This is justified by two assumptions: first, that every hypothesis in simultaneous editing must have at least one match in every record; and second, that all hypotheses will be represented as conjunctions of features. Given these two assumptions, only features that match somewhere in every record will be useful for constructing hypotheses. The region set matching each named pattern is also intersected with `in` *`record`*, where *`record`* is the set of records, in order to eliminate matches that lie outside the record set.

By similar reasoning, the only useful literal features are substrings that occur at least once in every record. The common substrings of the records are found using a suffix tree, the same way that `contains` features are found for selection guessing.

After generating library features and literal features, the features are sorted in order of preference. This step essentially determines the preference bias of the learning algorithm. Placing the most-preferred features first makes the hypothesis search simpler. When the features are ordered, the search can just scan down the list of features and stop as soon as it finds the feature it needs to satisfy the examples, since this feature is guaranteed to be the most preferred consistent feature.

Features are classified into three groups for the purpose of preference ordering:

- *unique features,* which occur exactly once in each record;
- *regular features,* which occur exactly n times in each record, for some $n > 1$; and
- *varying features,* which occur a varying number of times in each record.

A feature's classification is not predetermined. Instead, it is found by counting occurrences in the records being edited. For example, in Figure 9.4, `Type` is unique because it occurs exactly once in every record. Regular features are commonly found as delimiters. For example, if the records are IP addresses like 127.0.0.1, then "." is a regular feature. Varying features are typically tokens like words and numbers, which are general enough to occur in every record but do not necessarily follow any regular pattern.

Unique features are preferred over the other two kinds. A unique feature has the simplest description: the feature name itself, like `Type`. By contrast, using a regular or varying feature in a hypothesis requires specifying the index of a particular occurrence, such as `5th Word`. Regular features are preferred over varying features, because regularity of occurrence is a strong indication that the feature is relevant to the internal structure of a record.

Within each kind of feature, library features are preferred over literal features. Among literal features, longer literals are preferred to shorter ones. Among library features, however, no ordering is currently defined. In the future, it would be useful to let the user specify preferences between patterns in the library, so that, for instance, Java features could be preferred over character-class features.

To summarize, feature generation orders the feature list in the following order, with most preferred features listed first:

1. unique library patterns
2. unique literals
3. regular library patterns
4. regular literals
5. varying library patterns
6. varying literals

Within each group of library patterns, the order is arbitrary. Within each group of literals, longer literals are preferred to shorter.

Hypothesis Generation

Hypothesis generation takes the user's positive examples and the feature list, and attempts to generate a region set consistent with the examples. Unlike the selection guessing algorithm, however, only one hypothesis is returned by this search. This decision was made so that the hypothesis search would be as fast as possible, allowing simultaneous editing to have the best possible response time.

The search process works as follows. Using the first positive example, called the *seed example*, the system scans through the feature list, testing the seed example for membership in each feature's region set. When a matching feature `F` is found, the system constructs one or more candidate descriptions representing the particular occurrence that matched, depending on the type of the feature:

- if `F` is a unique feature, then the candidate description is just `F`.
- if `F` is a regular feature, then the candidate description is either `ith F` or `jth from last F`, whichever index is smaller.
- if `F` is a varying feature, then two candidate descriptions are generated: `ith F` and `jth from last F`.

These candidate descriptions are tested against the other positive examples, if any. The first consistent description found is returned as the hypothesis.

The output of the search process depends on whether the user is placing an insertion point (e.g. by clicking the mouse) or selecting a region (e.g. by dragging). If all the positive examples are zero-length regions, then the system assumes that the user is placing an insertion point, and searches for a point description. Otherwise, the system searches for a region description.

To search for a point description, the system transforms the seed example, which is just a character offset b, into two region rectangles: $(b, b, b, +\infty)$, which represents all regions that start at b, and $(-\infty, b; b, b)$, which represents all regions that end at b. The search then tests these region rectangles for intersection with each feature in the feature list. Candidate descriptions generated by the search are transformed into point descriptions by prefixing `point just before` or `point just after`, depending on which region rectangle matched the feature, and then the descriptions are tested for consistency with the other positive examples.

To search for a region description, the system first searches for the seed example using the basic search process described above. If no matching feature is found — because the seed example does not correspond precisely to a feature on the feature list — then the system splits the seed example into its start point and end point, and recursively searches for point descriptions for each point. Candidate descriptions for the start point and end point are transformed into a description of the entire region by wrapping them with `from...to...`, and then tested for consistency with the other examples.

This search algorithm is capable of generalizing a selection only if it starts and ends on a feature boundary. For literal features, this is not constraining at all. Since a literal feature is a string that occurs in all records, every substring of a literal feature is *also* a literal feature. Thus every position in a literal feature lies on a feature boundary. To save space, only maximal literal features are stored in the feature list, and the search phase tests whether the seed example falls anywhere inside a maximal literal feature.

Update

In simultaneous editing, the user is not only making selections, but also editing the document. Editing has two effects on inference. First, every edit changes the start and end offsets of regions, so the region sets used to represent features become incorrect. Second, editing changes the document

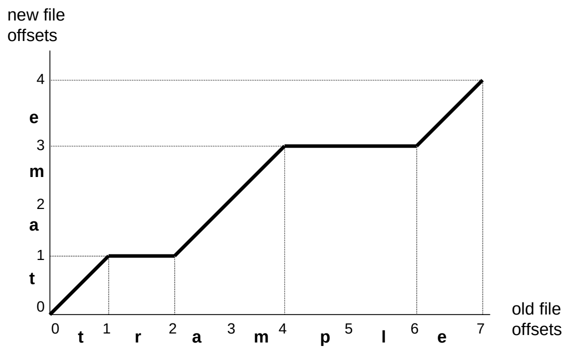


Figure 9.16: Coordinate map translating offsets between two versions of a document. The old version is the word `trample`. Two regions are deleted to get the new version, `tame`.

content, so the precomputed features may become incomplete or wrong. For example, after the user types some new words, the precomputed `Word` feature becomes incomplete, since it doesn't include the new words the user typed. The update algorithm addresses these two problems.

From the locations and length of text inserted or deleted, the updating algorithm computes a *coordinate map*, a relation that translates a character offset prior to the change into the equivalent character offset after the change. (Coordinate maps were described in Section 6.2.12.) The coordinate map can translate coordinates in either direction. For example, Figure 9.16 shows the coordinate map for a simple edit. Offset 3 in `trample` corresponds to offset 2 in `tame`, and vice versa. Offsets with more than one possible mapping in the other version, such as offset 1 in `tame`, are resolved arbitrarily; LAPIS picks the largest value.

Since the coordinate map for a group of insertions or deletions is always piecewise linear, it is represented as a sorted list of the endpoints of each line segment. If a single edit consists of m insertions or deletions (one for each record), then this representation takes $O(m)$ space. Evaluating the coordinate map function for a single offset takes $O(\log m)$ time using binary search.

A straightforward way to use the coordinate map is to scan down the feature list and update the start and end points of every feature to reflect the change. If the feature list is long, however, and includes some large feature sets like `Word` or `Whitespace`, the cost of updating every feature after every edit may be prohibitive. LAPIS takes the reverse strategy: instead of translating all features up to the present, the user's examples are translated back to the past. The system maintains a global coordinate map representing the translation between original document coordinates when the feature list was generated and the current document coordinates. When an edit occurs, the updating algorithm computes a coordinate map for the edit and composes it with this global coordinate map. When the user provides examples for a new selection, the examples are translated back to the original document coordinates using the inverse of the global coordinate map. The search algorithm generates a consistent description for the translated examples. The generated description is then translated forward to current document coordinates before being displayed as a selection.

An important design decision in a simultaneous editing system that uses domain knowledge, such as Java syntax, is whether the system should attempt to reparse the document while the user is editing it. On one hand, reparsing would allow the system to track all the user's changes and reflect those changes in its descriptions. On the other hand, reparsing is expensive and may fail if the document is in an intermediate, syntactically incorrect state. LAPIS never reparses automatically in simultaneous editing mode. The user can explicitly request reparsing with a menu command, which effectively restarts simultaneous editing using the same set of records. Otherwise, the feature list remains frozen in the original version of the document. One consequence of this decision is that the human-readable descriptions returned by the inference algorithm may be misleading because they refer to an earlier version.

This design decision raises an important question. If the feature list is frozen, how can the user make selections in newly-inserted text, which didn't exist when the feature list was built? This problem is handled by the update algorithm. Every character typed in simultaneous editing mode adds a new literal feature to the feature list, since typed characters are guaranteed to be identical in all the records. Similarly, pasting text from the clipboard creates a special feature that translates coordinates back to the source of the paste and tries to find a description there. When a hypothesis uses one of these features created by editing, the feature is described as "somewhere in edit$N$", which can be seen in Figure 9.9.

A disadvantage of this scheme is that the housekeeping structures – the global coordinate map and the new features added for edits – grow steadily as the user edits. This growth can be slowed significantly by coalescing adjacent insertions and deletions, although LAPIS does not yet include this optimization. Another solution might be to reparse when the number of edits reaches some threshold, doing the reparsing in the background on a copy of the document in order to avoid interfering with the user's editing. In practice, however, space growth doesn't seem to be a serious problem. For most tasks, the user spends only a few minutes in a simultaneous editing session, not the hours that are typical of general text editing. After leaving simultaneous editing mode, the global coordinate map and the feature list can be discarded.

9.3 User Studies

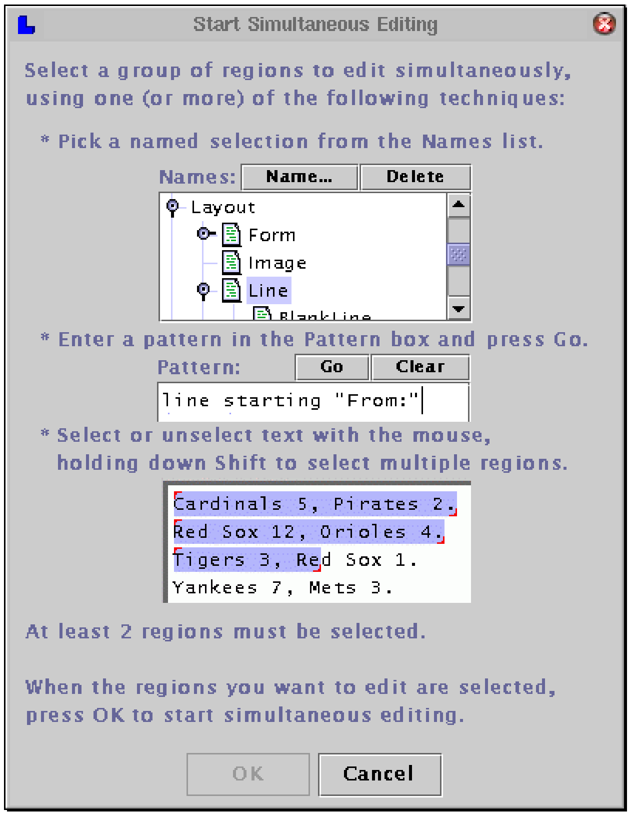
Selection guessing and simultaneous editing were tested with a pair of user studies, described in this section.

The first study evaluated simultaneous editing, which was actually the first of the two selection inference modes to be designed and implemented. At the time the study was done, the user interface for simultaneous editing was somewhat different. The most significant difference affected the first step of simultaneous editing, in which the user selects the records. Because selection guessing had not been implemented yet, the records had to be selected a different way, by choosing a named pattern from the library pane. As a result, the dialog boxes were worded differently. Screenshots of the original dialog boxes are included in the description of the study below.

The second study evaluated selection guessing, using one of the tasks used in the simultaneous editing task, so that some comparisons can be made between the two techniques.

9.3.1 Simultaneous Editing Study

The first study was designed to evaluate the usability of simultaneous editing on small repetitive editing tasks. The study compared simultaneous editing with manual (single-cursor) editing along two quantitative dimensions: time to complete the task, and errors in the finished product. Manual



Select a group of regions to edit simultaneously, using one (or more) of the following techniques:

* Pick a named selection from the Names list.

* Enter a pattern in the Pattern box and press Go.

line starting "From:"

* Select or unselect text with the mouse, holding down Shift to select multiple regions.

```
Cardinals 5, Pirates 2.
Red Sox 12, Orioles 4.
Tigers 3, Red Sox 1.
Yankees 7, Mets 3.
```

At least 2 regions must be selected.

When the regions you want to edit are selected, press OK to start simultaneous editing.

Figure 9.17: The Simultaneous Editing dialog box used in the user study.

editing was chosen as the basis for comparison because the users were already experts in manual editing, and manual editing would often be the natural alternative for these small tasks. Still, it is also desirable to compare simultaneous editing with other research approaches. Two of the tasks used in the study were borrowed from the user study of another programming-by-demonstration system, DEED [Fuj98], allowing the performance of simultaneous editing on those tasks to be compared with the performance of DEED.

In the version of LAPIS tested in this study, entering simultaneous editing mode popped up a dialog box window instead of a dialog pane. This dialog box (Figure 9.17) prompted the user to select the record set using one of the existing selection methods — mouse, pattern, or library. For all the tasks in the user study, the record set could be selected using a library pattern (either `Line` or `Paragraph`), so it was not necessary to teach users the TC pattern language.

Procedure

Eight users were found by soliciting campus job newsgroups (`cmu.misc.jobs` and `cmu.misc.market`). All were college undergraduates with substantial text-editing experience and varying levels of programming experience (5 described their programming experience as "little" or "none," and 3 as "some" or "lots"). Users were paid $5 for participation.

Users first learned about simultaneous editing by reading a tutorial and trying the examples. The tutorial led the user through two example tasks, showing step by step how the tasks could be performed with simultaneous editing. The tutorial tasks are shown in Figures 9.18 and 9.19. The

1. Let a set of state variables R be null. Let a sequence of pieces of input s contain only the goal input. Let a piece of input i be the goal input.
2. Add state variables vj's in REF(i) to R, if not included already.
3. If R is null, end the phase.
4. Search the piece of input p that immediately precedes i in the source input.
5. If p is not found, i.e., the search reaches the beginning of the source input, add the pieces of input that initialize vj's in R to the head of s, and end the phase.
6. If C = DEF(p) * R is not null, add p to the head of s, remove vj's in C from R, let i be p, and go to 2.
7. Let i be p, and go to 4.

↓

(1) Let a set of state variables R be null. Let a sequence of pieces of input s contain only the goal input. Let a piece of input i be the goal input.
(2) Add state variables vj's in REF(i) to R, if not included already.
(3) If R is null, end the phase.
(4) Search the piece of input p that immediately precedes i in the source input.
(5) If p is not found, i.e., the search reaches the beginning of the source input, add the pieces of input that initialize vj's in R to the head of s, and end the phase.
(6) If C = DEF(p) * R is not null, add p to the head of s, remove vj's in C from R, let i be p, and go to 2.
(7) Let i be p, and go to 4.

Figure 9.18: Tutorial task 1: change the numbering of a list.

tutorial part lasted less than 10 minutes for all but one user, who spent 30 minutes exploring the system.

After the tutorial, each user performed three tasks with simultaneous editing. The three tasks are shown in their entirety in Figures 9.20–9.22. All tasks were obtained from other authors: tasks 1 and 2 from Fujishima [Fuj98] and task 3 from Nix [Nix85]. After performing a task with simultaneous editing, users repeated the same task in manual selection mode, but only on the first three records to avoid unnecessary tedium. Users were instructed to work carefully and accurately at their own pace. All users were satisfied that they had completed all tasks, although the finished product sometimes contained undetected errors, a problem discussed further below.

LAPIS was instrumented to capture all selections made by the user, all inferences made by the system, and all editing actions. An experimenter also observed all the sessions and made notes. No audio or video recordings were made.

Results

Aggregate times for each task are shown in Table 9.1. No performance differences were seen between programmers and nonprogrammers.

Following the analysis used by Fujishima [Fuj98], the leverage obtained with simultaneous editing can be estimated by dividing the time to edit all records with simultaneous editing by the time to edit just one record manually. This ratio, which I call *equivalent task size*, represents the number of records for which simultaneous editing time would be equal to manual editing time

Date: 01/10/97
Sender: Taro Yamada
Subject: Workshop Info

Date: 01/14/97
Sender: Hanako Takada
Subject: Lunch

Date: 01/15/97
Sender: Takashi Takahashi
Subject: Agenda

Date: 01/17/97
Sender: Kazuo Tanaka
Subject: Internet & AI

Date: 01/17/97
Sender: Yuzo Fujishima
Subject: Paper Review

↓

Workshop Info (Taro)
Lunch (Hanako)
Agenda (Takashi)
Internet & AI (Kazuo)
Paper Review (Yuzo)

Figure 9.19: Tutorial task 2: reformat email message headers into a short list.

Task	Simultaneous editing	Manual editing
1	142.9 s [63–236 s]	21.6 s/rec [7.7–65 s/rec]
2	119.1 s [64–209 s]	32.3 s/rec [19–40 s/rec]
3	159.6 s [84–370 s]	41.3 s/rec [16–77 s/rec]

Table 9.1: Time taken by users to perform each task (mean [min–max]). *Simultaneous editing* is the time to do the entire task with simultaneous editing. *Manual editing* is the time to edit 3 records of the task by hand, divided by 3 to get a per-record estimate.

	Simultaneous editing		DEED
Task	novices	expert	novices
1	8.4 recs [2.1–12.2 recs]	4.5 recs	67 recs [6.5–220 recs]
2	3.6 recs [1.9–5.8 recs]	1.6 recs	28 recs [7–130 recs]
3	4.0 recs [1.9–6.2 recs]	2.4 recs	

Table 9.2: Equivalent task sizes for each task (mean [min–max]). *Novices* are users in the user study. *Expert* is my own performance, provided for comparison. *DEED* is another PBD system, tested on tasks 1 and 2 by its author [Fuj98].

1. Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799.
2. Brajnik, G. and Tasso, C. A Shell for developing non-monotonic user modeling systems. Int. J. Human-Computer Studies 40 (1994), 31-62.
3. Cohen, P.R., Cheyer, A., Wang, M., and Baeg, S.C. An Open Agent Architecture. In Software Agents: Papers from the AAAI 1994 Spring Symposium, AAAI Press, 1994, pp. 1-8.
4. Corkill, D.D. Blackboard Systems. AI Expert 6, 9 (Sep. 1991), 40-47.
5. Cranefield, S. and Purvis, M. Agent-based Integration of General Purpose Tools. In Proceedings of the Workshop on Intelligent Information Agents. Fourth International Conference on Information and Knowledge Management, Baltimore, 1995.
6. Finin, T, Fritzson, R., McKay, D. and McEntire, R. KQML A Language and Protocol for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
7. Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass., 1993.
9. Martin, D.L., Cheyer, A., and Lee, G-L. Development Tools for the Open Agent Architecture. In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology. The Practical Application Company Ltd., London, April 1996, pp. 387-404.

↓

[Aha 89] Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799.
[Brajnik 94] Brajnik, G. and Tasso, C. A Shell for developing non-monotonic user modeling systems. Int. J. Human-Computer Studies 40 (1994), 31-62.
[Cohen 94] Cohen, P.R., Cheyer, A., Wang, M., and Baeg, S.C. An Open Agent Architecture. In Software Agents: Papers from the AAAI 1994 Spring Symposium, AAAI Press, 1994, pp. 1-8.
[Corkill 91] Corkill, D.D. Blackboard Systems. AI Expert 6, 9 (Sep. 1991), 40-47.
[Cranefield 95] Cranefield, S. and Purvis, M. Agent-based Integration of General Purpose Tools. In Proceedings of the Workshop on Intelligent Information Agents. Fourth International Conference on Information and Knowledge Management, Baltimore, 1995.
[Finin 94] Finin, T, Fritzson, R., McKay, D. and McEntire, R. KQML A Language and Protocol for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowledge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
[Hayes 95] Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
[Kosbie 93] Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstration. The MIT Press, Cambridge, Mass., 1993.
[Martin 96] Martin, D.L., Cheyer, A., and Lee, G-L. Development Tools for the Open Agent Architecture. In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology. The Practical Application Company Ltd., London, April 1996, pp. 387-404.

Figure 9.20: Task 1: put author name and year in front of each citation in a bibliography [Fuj98].

```
<DT><A HREF="mailto:cg@cs.umn.edu" NICKNAME="congra">Conceptual Graphs</A>
<DT><A HREF="mailto:DAI-List@ece.sc.edu" NICKNAME="dai">Distributed Artificial Intel-
ligence</A>
<DT><A HREF="mailto:ii_chi@acm.org" NICKNAME="ii">Intelligent Interfaces</A>
<DT><A HREF="mailto:kif@cs.stanford.edu" NICKNAME="kif">KIF</A>
<DT><A HREF="mailto:kqml@cs.umbc.edu" NICKNAME="kqml">KQML</A>
<DT><A HREF="mailto:agents@cs.umbc.edu" NICKNAME="swagent">Software Agents</A>
<DT><A HREF="mailto:tscript@brownvm.brown.edu" NICKNAME="tscript">Telescript</A>
```

↓

```
;; Conceptual Graphs
congra: mailto:cg@cs.umn.edu
;; Distributed Artificial Intelligence
dai: mailto:DAI-List@ece.sc.edu
;; Intelligent Interfaces
ii: mailto:ii_chi@acm.org
;; KIF
kif: mailto:kif@cs.stanford.edu
;; KQML
kqml: mailto:kqml@cs.umbc.edu
;; Software Agents
swagent: mailto:agents@cs.umbc.edu
;; Telescript
tscript: mailto:tscript@brownvm.brown.edu
```

Figure 9.21: Task 2: reformat a list of mail aliases from HTML to plain text [Fuj98].

```
Cardinals 5, Pirates 2.
Red Sox 12, Orioles 4.
Tigers 3, Red Sox 1.
Yankees 7, Mets 3.
Dodgers 6, Tigers 4.
Brewers 9, Braves 3.
Phillies 2, Reds 1.
```

↓

```
GameScore[winner ’Cardinals’; loser ’Pirates’; scores[5, 2]].
GameScore[winner ’Red Sox’; loser ’Orioles’; scores[12, 4]].
GameScore[winner ’Tigers’; loser ’Red Sox’; scores[3, 1]].
GameScore[winner ’Yankees’; loser ’Mets’; scores[7, 3]].
GameScore[winner ’Dodgers’; loser ’Tigers’; scores[6, 4]].
GameScore[winner ’Brewers’; loser ’Braves’; scores[9, 3]].
GameScore[winner ’Phillies’; loser ’Reds’; scores[2, 1]].
```

Figure 9.22: Task 3: reformat a list of baseball scores into a tagged format [Nix85].

for a given user. Since manual editing time increases linearly with record number, and simultaneous editing time is roughly constant or only slowly increasing, simultaneous editing will be faster whenever the number of records is greater than the equivalent task size. Note that the average equivalent task size is not necessarily equal to the ratio of the average editing times, since $E[S/M] \neq E[S]/E[M]$.

As Table 9.2 shows, the average equivalent task sizes are small. In other words, the average novice user works faster with simultaneous editing if there are more than 8.4 records in the first task, more than 3.6 records in the second task, or more than 4 records in the third task.[2] Thus simultaneous editing is an improvement over manual editing even for very small repetitive editing tasks, and even for users with as little as 10 minutes of experience. Some users were so slow at manual editing that their equivalent task size is smaller than the expert's, so simultaneous editing benefits them even more.

Simultaneous editing also compares favorably to another PBD system, DEED [Fuj98]. When DEED was evaluated with novice users on tasks 1 and 2, the reported equivalent task sizes averaged 67 for task 1 and 28 for task 2, roughly an order of magnitude larger than simultaneous editing. The variability of equivalent task sizes across users was also considerably greater for DEED.

Another important aspect of system performance is inference accuracy. Each incorrect inference forces the user to make at least one additional action, such as selecting a counterexample or providing an additional positive or negative example. In the user study, users made a total of 188 selections that were used for editing. Of these, 158 selections (84%) were correct immediately, requiring no further examples. The remaining selections needed either 1 or 2 extra examples to generalize correctly. On average, only 0.26 additional examples were needed per selection.

Unfortunately, users tended to overlook slightly-incorrect inferences, particularly inferences that selected only half of the hyphenated author "Hayes-Roth" or the two-word baseball team "Red Sox". As a result, the overall error rate for simultaneous editing was slightly worse than for manual editing: 8 of the 24 simultaneous editing sessions ended with at least one uncorrected error, whereas 5 of 24 manual editing sessions ended with uncorrected errors. If the two most common selection errors had been noticed by users, the error rate for simultaneous editing would have dropped to only 2 of 24. These observations led to the idea of *outlier finding*, which is covered in Chapter 10.

After doing the tasks, users were asked to evaluate the system's ease-of-use, trustworthiness, and usefulness on a 5-point Likert scale. These questions were also borrowed from Fujishima [Fuj98]. The results, shown in Figure 9.23, are generally positive.

9.3.2 Selection Guessing Study

The second study was designed to evaluate the usability of selection guessing on repetitive editing tasks. In order to compare selection guessing with simultaneous editing, the overall design of the study was the same as the first study, except that users used selection guessing to perform tasks. Unfortunately, not all the tasks of the first study can be performed with selection guessing. Tasks 1 and 3 require counted patterns for some of their selections (e.g., `first CapitalizedWord`

[2]These estimates are actually conservative. Simultaneous editing always preceded manual editing for each task, so the measured time for simultaneous editing includes time spent thinking about and understanding the task. For the manual editing part, users had already learned the task, and were able to edit at full speed.

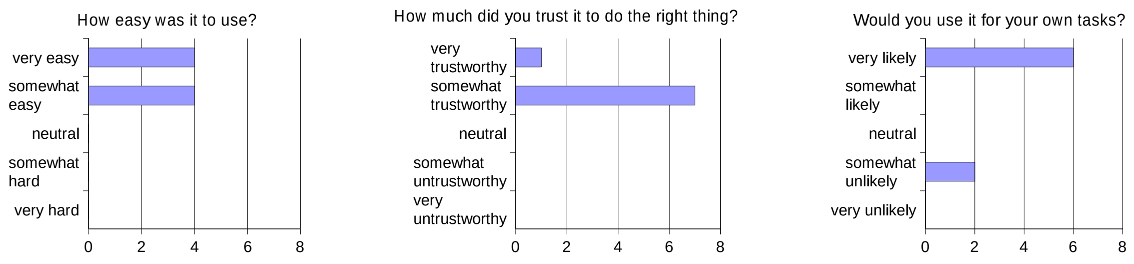
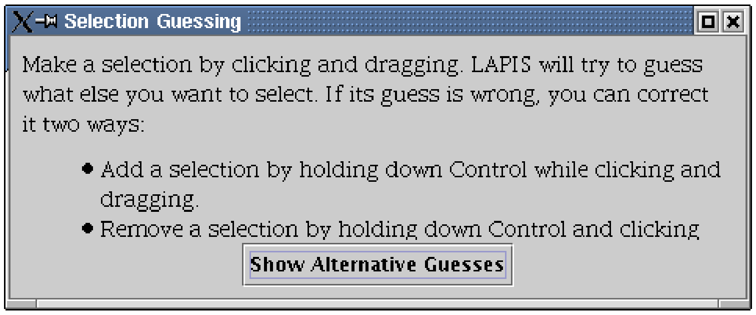


Figure 9.23: User responses to questions about simultaneous editing.



Make a selection by clicking and dragging. LAPIS will try to guess what else you want to select. If its guess is wrong, you can correct it two ways:

- Add a selection by holding down Control while clicking and dragging.
- Remove a selection by holding down Control and clicking

Figure 9.24: The Selection Guessing dialog box used in the user study.

`in Line`). These patterns can be inferred by the simultaneous editing algorithm, but not by the selection guessing algorithm. As a result, the selection guessing study used only task 2.

As was the case in the previous study, the user interface of selection guessing mode was slightly different when the study was performed. The selection guessing dialog was a popup window rather than a pane, and the controls were arranged somewhat differently, although all the same features were available. Figure 9.24 shows the dialog box used in the study.

Procedure

Five users were found by soliciting campus job newsgroups (`cmu.misc.jobs` and `cmu.misc.market`). All were college undergraduates with substantial text-editing experience and varying levels of programming experience (1 described their programming experience as "little," 3 as "some," and 1 as "lots"). Users were paid $5 for participation.

As in the previous study, users learned about selection guessing by reading a tutorial and trying the examples. The tutorial led the user through one example task (Figure 9.19). The tutorial part lasted less than 10 minutes for all users.

After the tutorial, each user performed one task in selection guessing mode (Task 2, Figure 9.21), and then repeated the first three records of the task in manual selection mode. Users were instructed to work carefully and accurately at their own pace.

Results

Four out of five users were able to complete the task entirely with selection guessing. The fifth user also completed the task, but only by exiting selection guessing mode at one point, doing a troublesome part with manual editing, and then resuming selection guessing to finish the task.

Task	Selection guessing	Manual editing
2	426.0 s [173–653 s]	43.0 s/rec [32–52 s/rec]

Table 9.3: Time taken by users to perform the task (mean [min–max]). *Selection guessing* is the time to do the entire task with selection guessing. *Manual editing* is the time to edit 3 records of the task by hand, divided by 3 to get a per-record estimate.

	Selection guessing		Simultaneous editing		DEED
Task	novices	expert	novices	expert	novices
2	9.3 recs [4.7–15.7 recs]	2.3 recs	3.6 recs [1.9–5.8 recs]	1.6 recs	28 recs [7–130 recs]

Table 9.4: Equivalent task sizes (mean [min–max]) for task 2, comparing selection guessing, simultaneous editing, and DEED. *Novices* are users in a user study. *Expert* is my own performance, provided for comparison.

Aggregate times for selection guessing and manual editing are shown in Table 9.3. The times for selection guessing include the detour into manual editing made by one user.

The same analysis as the first study is used to compute the equivalent task size for doing this task with selection guessing. The results are shown in Table 9.4. For comparison, the table also includes the equivalent task sizes for the same task in the simultaneous editing study and the DEED study. Selection guessing is not as fast as simultaneous editing on this task, but still outperforms DEED.

One reason that selection guessing was slower is less accurate inference. Of the 51 selections used for editing in the selection guessing study, only 34 (67%) were inferred correctly from one example. By comparison, in the simultaneous editing study, *all* selections on task 2 were inferred correctly from one example.

In selection guessing, an incorrect inference can be corrected in three ways: giving another positive example, giving a negative example, or selecting an alternative hypothesis. To judge the user cost of inference, then, we must measure the number of *actions* needed to create a selection, where an action is either giving an example or clicking on an alternative hypothesis. On average, 2.73 actions were needed to create each selection used for editing in selection guessing, compared to 1 action per selection in simultaneous editing.

After the study, users evaluated selection guessing's ease-of-use, trustworthiness, and usefulness on a 5-point Likert scale. The results, shown in Figure 9.25, are considerably more mixed than for simultaneous editing.

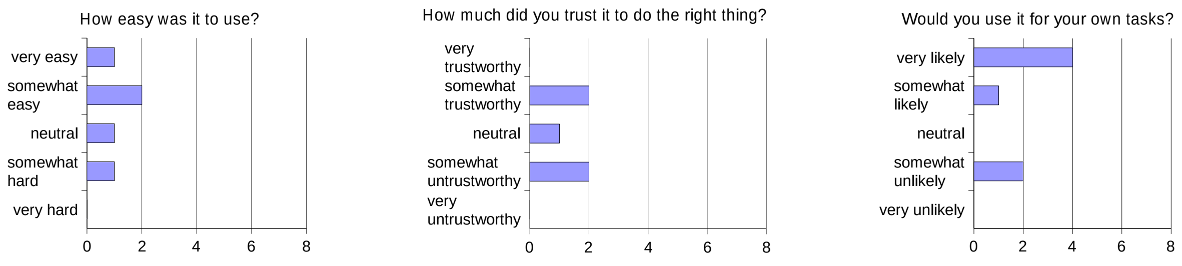


Figure 9.25: User responses to questions about selection guessing.

Chapter 10

Outlier Finding

Testing and debugging are important problems for any kind of automation, and pattern matching is no exception. Users may write incorrect patterns that inadvertently match some regions that were not meant to be matched (*false positives*) or fail to match some regions that should have been matched (*false negatives*). Inferred patterns may also need debugging. Even though 84% of inferred selections in the simultaneous-editing study (Section 9.3.1) needed only one example, the remaining 16% were *wrong* after the first example. Users have to notice that an inferred selection is incorrect before they can fix it by giving additional examples. Unfortunately, as noted in the discussion of that study, users did not always notice incorrect inferences, and made errors as a result.

This chapter[1] presents a new technique for debugging pattern matches: *outlier finding*. In statistics, an outlier is a data point which appears to be inconsistent with the rest of the data [BL84]. Applied to text pattern matching, an outlier is a pattern match (or mismatch) which differs significantly from other matches (or mismatches). Automatic outlier finding helps the user focus attention on the cases that are most likely to be problematic.

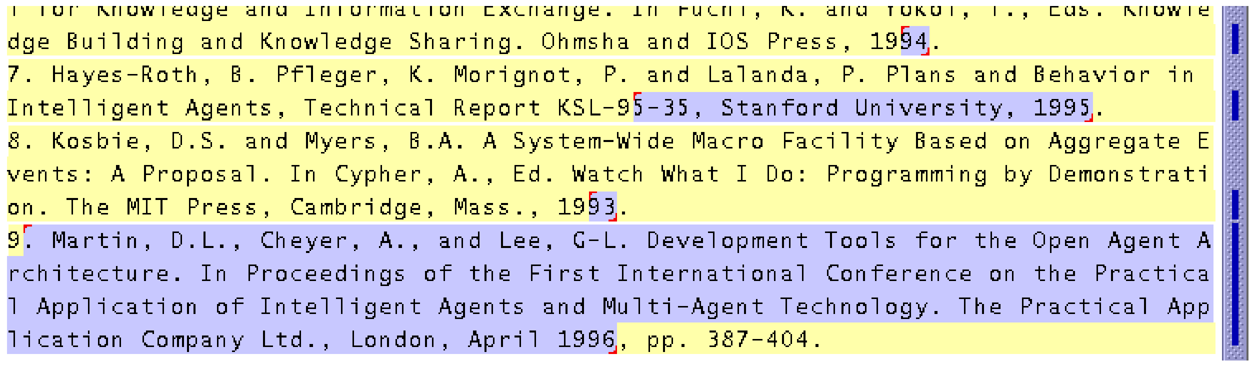
Briefly, the outlier finding algorithm takes any region set, generates a vector of binary-valued features representing each region, and then sorts the regions based on their weighted Euclidean distance from the median in feature vector space. Regions that lie far from the median are considered outliers.

In the LAPIS user interface, outlier finding is used in two ways. First, during selection inference, outliers in the system's inference are highlighted automatically to draw the user's attention to them. Second, in any mode, the user can use an "Unusual Matches" visualization to view and explore the currently-selected regions in outlier order.

Outlier finding depends on two assumptions. First, most matches must be correct, so that errors are the needles in the haystack, not the hay. This assumption is essential because the outlier finder has no way of knowing what region set the user actually intends the pattern to match. Unless the region set is roughly correct to begin with, the outlier finder's suggestions are unlikely to be helpful. Fortunately, outlier finding has more value when errors are rare, since it means the errors are harder for the user to find by a manual search.

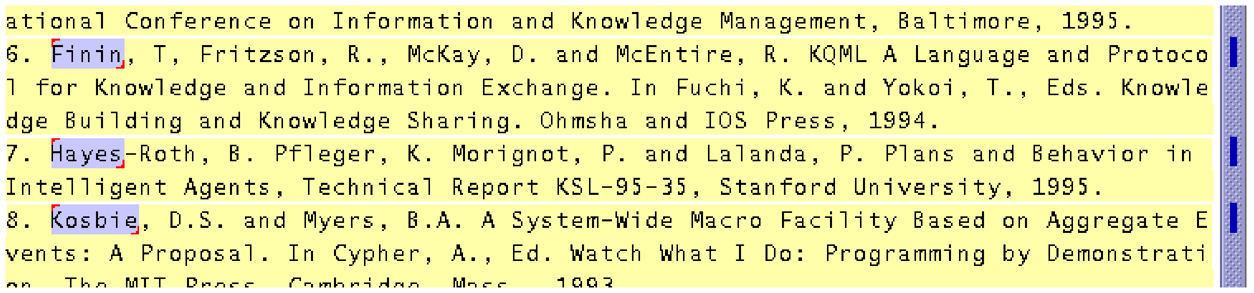
Second, erroneous matches must differ from correct matches in ways that can be captured by the features available to the outlier finder. Although the outlier finder uses all the domain

[1]Portions of this chapter are adapted from an earlier paper [MM01b].



```
l for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowle
dge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
7. Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in
Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate E
vents: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstrati
on. The MIT Press, Cambridge, Mass., 1993.
9. Martin, D.L., Cheyer, A., and Lee, G-L. Development Tools for the Open Agent A
rchitecture. In Proceedings of the First International Conference on the Practica
l Application of Intelligent Agents and Multi-Agent Technology. The Practical App
lication Company Ltd., London, April 1996, pp. 387-404.
```

Figure 10.1: Incorrect inference of the last two digits of the publication year. The selection is visibly wrong, and all users noticed and corrected it.



```
ational Conference on Information and Knowledge Management, Baltimore, 1995.
6. Finin, T, Fritzson, R., McKay, D. and McEntire, R. KQML A Language and Protoco
l for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowle
dge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
7. Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in
Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate E
vents: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstrati
on. The MIT Press, Cambridge, Mass., 1993
```

Figure 10.2: Incorrect inference of the author's name. "Hayes-Roth" is only partially selected, but no users noticed.

knowledge found in the LAPIS pattern library, the knowledge base inevitably has gaps, and the feature language may be incomplete. Any learning system has this problem, since it is impossible to infer a concept that cannot be represented. Fortunately, the LAPIS pattern library is easy to extend.

The next section delves into the kinds of errors made by users in the simultaneous editing study, in order to better motivate the outlier-finding solution. Subsequent sections describe outlier highlighting and the Unusual Matches dialog, which are the user interface techniques in LAPIS that rely on outlier finding. The fourth section details the outlier finding algorithm. The chapter concludes with a user study of outlier highlighting.

10.1 Motivation

Observations from the simultaneous-editing user study showed that some incorrect inferences are far more noticeable than others. Figure 10.1 shows an incorrect inference that was easy for users to notice. The user has selected "89", the last two digits of the first record's publication year, from which the system has inferred the description `from end of first "9" to end of first year`. This inference is drastically, visibly wrong, selecting far more than two digits in some records and nearly the entire last record. All eight users in the study noticed this error and corrected it by giving another example.

The mistake in Figure 10.2, on the other hand, was much harder to spot. The user has selected the last name of the first author, “Aha”. The system’s inference is `1st CapitalizedWord`, which is correct for all but record 7, where it selects only the first half of the hyphenated name “Hayes-Roth”. The error is so visually subtle that all seven users who made this selection or a related selection completely overlooked the error and used the incorrect selection anyway. (The eighth user luckily avoided the problem by including the comma in the selection, which was inferred correctly.) Although three users later noticed the mistake and managed to change “[Hayes 95]” to the desired “[Hayes-Roth 95]”, the other four users never noticed the error at all. A similar effect was seen in another task, when some users failed to notice that the two-word baseball team “Red Sox” was not selected correctly.

The difference between an inference whose errors were noticed by *all* users, and another whose errors were noticed by *none*, is telling. The selection errors in Figure 10.1 are too prominent for users to ignore, but the errors in Figure 10.2 are too subtle to be noticed.

10.2 Outlier Highlighting

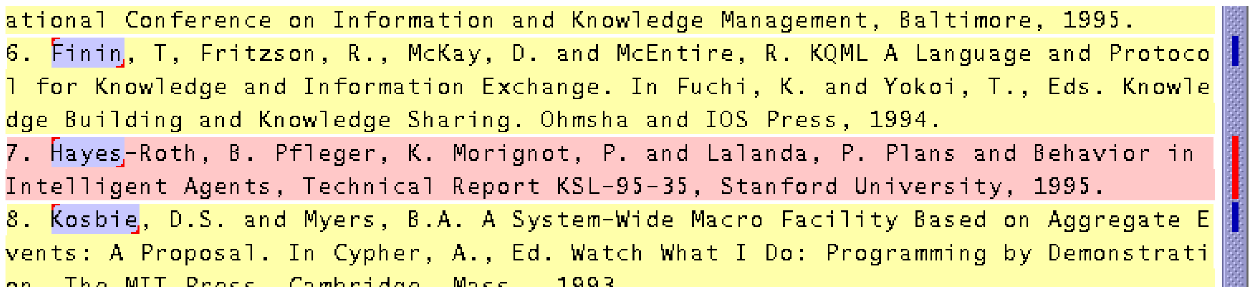
In an effort to make incorrect selections like Figure 10.2 more noticeable, outlier highlighting was added to LAPIS. Whenever the system makes an inference in either selection guessing mode or simultaneous editing mode, it passes the resulting set of selected regions to the outlier finder, which rank the regions by their distance from the average selection. Using this ranking, the system highlights the most unusual regions in a visually distinctive fashion, in order to attract the user’s attention so that they can be checked for errors.

Two design questions immediately arise: how many outliers should be highlighted, and how should they be highlighted? Outliers are not guaranteed to be errors. Highlighting too many outliers when the selection is actually correct may lead the user to distrust the highlighting hint. On the other hand, highlighting more outliers means more actual errors may be highlighted. But highlighting a large number of outliers is unhelpful to the user, since the user must examine each one. Ideally, the outlier finder should highlight only a handful of selections when the selection is likely to have errors, and none at all if the selection is likely to be correct.

The following heuristic seems to work well. Let d be the distance of the farthest selection from the average, and let S be the set of selections that are farther than $d/2$ from the average. If S is small – containing fewer than 10 selections or fewer than half of all the selections, whichever is smaller – then highlight every member of S as a outlier. Otherwise, do not highlight any selections as outliers. This algorithm puts a fixed upper bound on the number of outlier highlights, but avoids displaying useless highlights when the selections are not significantly different from one another.

The second design decision is how outlier highlighting should be rendered in the display. One possibility is the way Microsoft Word indicates spelling and grammar errors, a jagged, brightly colored underline. Experienced Word users are accustomed to this convention and already understand that it’s merely a hint. In LAPIS, however, outlier highlighting must stand out through selected text, which is rendered using a blue background. A jagged underline was found to be too subtle to be noticed in this context, particularly in peripheral vision.

Instead, an outlier selection is highlighted by changing its color from blue to red. In simultaneous editing mode, the highlighting is further enhanced by coloring the outlier’s entire record red. The scrollbar is also augmented with red marks corresponding to the highlighted outliers. LAPIS



```
ational Conference on Information and Knowledge Management, Baltimore, 1995.
6. Finin, T, Fritzson, R., McKay, D. and McEntire, R. KQML A Language and Protoco
l for Knowledge and Information Exchange. In Fuchi, K. and Yokoi, T., Eds. Knowle
dge Building and Knowledge Sharing. Ohmsha and IOS Press, 1994.
7. Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in
Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
8. Kosbie, D.S. and Myers, B.A. A System-Wide Macro Facility Based on Aggregate E
vents: A Proposal. In Cypher, A., Ed. Watch What I Do: Programming by Demonstrati
on. The MIT Press, Cambridge, Mass., 1993.
```

Figure 10.3: Incorrect inference with outlier highlighting drawing attention to the possible error.

already augments the scrollbar with marks corresponding to the selection (Section 7.3.4), so the red outlier marks are simply painted on top of the blue selection marks. Figure 10.3 shows the resulting display, highlighting an outlier in the erroneous author selection.

10.3 Unusual Matches Display

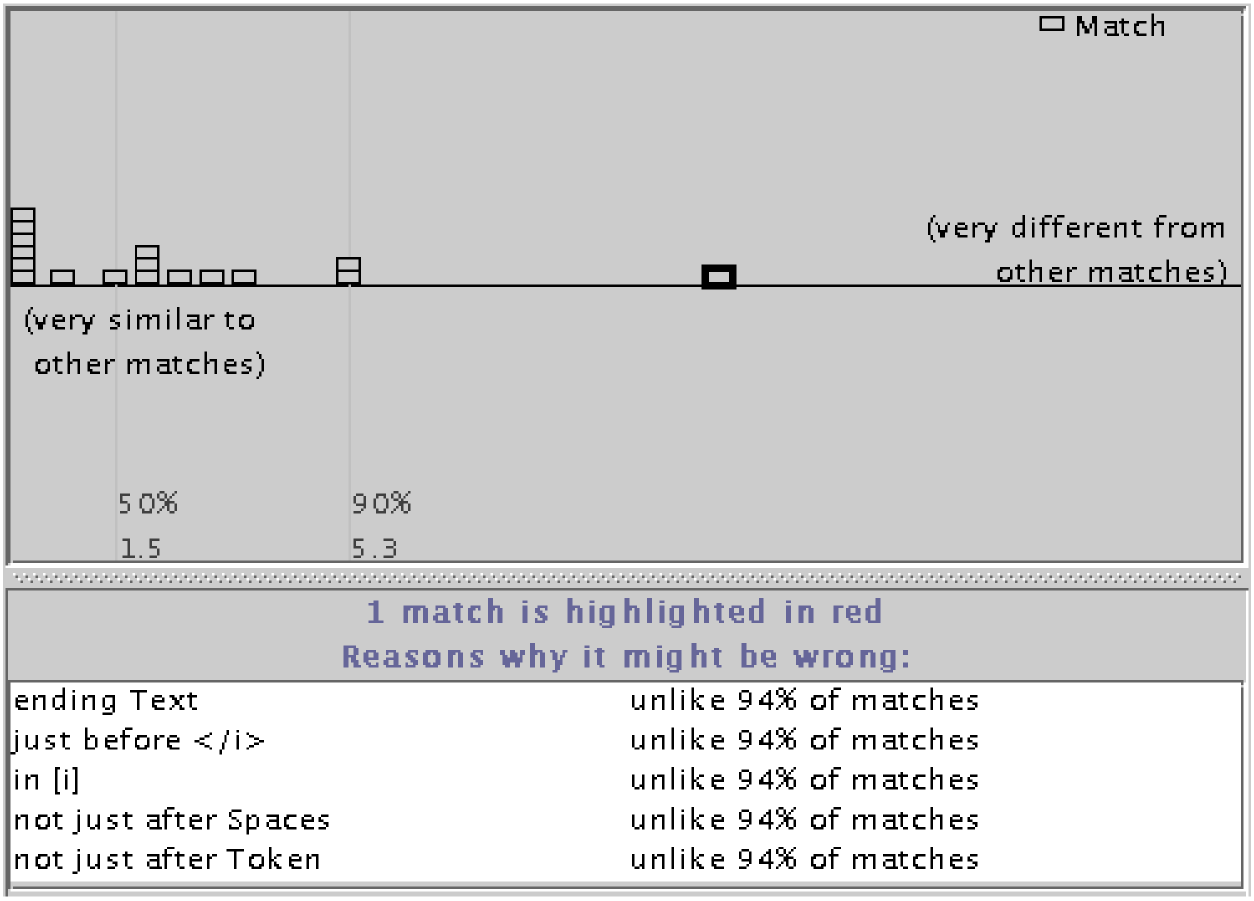
In simultaneous editing, outlier finding is used behind the scenes to direct the user's attention to possible errors. Some users may want to access the outlier finder directly, in order to explore the outliers and obtain explanations of each outlier's unusual features. For example, suppose a user is writing a pattern to replace all occurrences of a variable name in a large program, and the user wants to test and debug the pattern before using it. For this kind of task, LAPIS provides the Unusual Matches window (Figure 10.4).

The Unusual Matches window works in tandem with the LAPIS pattern matcher. Normally, when the user enters a pattern, LAPIS highlights all the pattern matches in the text editor. When the Unusual Matches window is showing, however, LAPIS also runs the outlier finder on the set of matches.

Unlike the outlier highlighting technique described in the previous section, the Unusual Matches window does not use a threshold to discriminate outliers from typical matches. Instead, it simply displays all the matches, in order of increasing *weirdness* (distance from the median), and lets the user decide which matches look like outliers. Each match is plotted as a small block. Blocks near the left side of the window represent typical matches, being very close to the median, and blocks near the right side represent outliers, far from the median. The distance between two adjacent blocks is proportional to their difference in weirdness. Strong outliers appear noticeably alone in this visualization (Figure 10.4).

Matches with identical feature vectors are combined into a cluster, shown as a vertical stack of blocks. Matches that lie at the same distance from the median in feature space, but along different vectors, are not combined into a stack. Instead, they are simply rendered side-by-side with 0 pixels between them.

The user can explore the matches by clicking on a block or stack of blocks, which highlights the corresponding regions in the text editor, using red highlights to distinguish them from the other pattern matches already highlighted in blue. The editor window scrolls automatically to display the highlighted region. If a stack of blocks was clicked, then the window scrolls to the first region in the stack and displays red marks in the scrollbar for the others. To go the other way, the user


▭ Match
(very different from other matches)
(very similar to other matches)
50%
1.5
90%
5.3

1 match is highlighted in red
Reasons why it might be wrong:

ending Text	unlike 94% of matches
just before </i>	unlike 94% of matches
in [i]	unlike 94% of matches
not just after Spaces	unlike 94% of matches
not just after Token	unlike 94% of matches

Figure 10.4: The Unusual Matches window showing occurrences of “copy” in a document. The most prominent outlier, which is selected, is found in an italicized word, “rcopy”.

can right-click on a selected region in the editing window and choose "Locate in Unusual Matches Window", which selects the corresponding block in the Unusual Matches window.

When a match is selected in the Unusual Matches window, the system also displays an explanation of how it is unusual (bottom pane in Figure 10.4). The explanation consists of the highest-weighted features (at most 5) in which the region differs from the median feature vector. If two features are related by generalization, such as `starts with Letters` and `starts with UpperCaseLetters`, only the higher-weighted feature is included in the explanation. Next to each feature in the explanation, the system displays the fraction of matches that agree with the median value — a statistic which is related to the feature's weight, but easier for the user to understand. The explanation generator is still rudimentary, and its explanations sometimes include obscure or apparently-redundant features. Generating better explanations is a hard problem for future work.

The Unusual Matches window can also show mismatches in the same display (Figure 10.5). When mismatches are displayed, the user can search for both kinds of bugs in a pattern: *false negatives* (mismatches which should be matches) as well as *false positives* (matches which should not be).

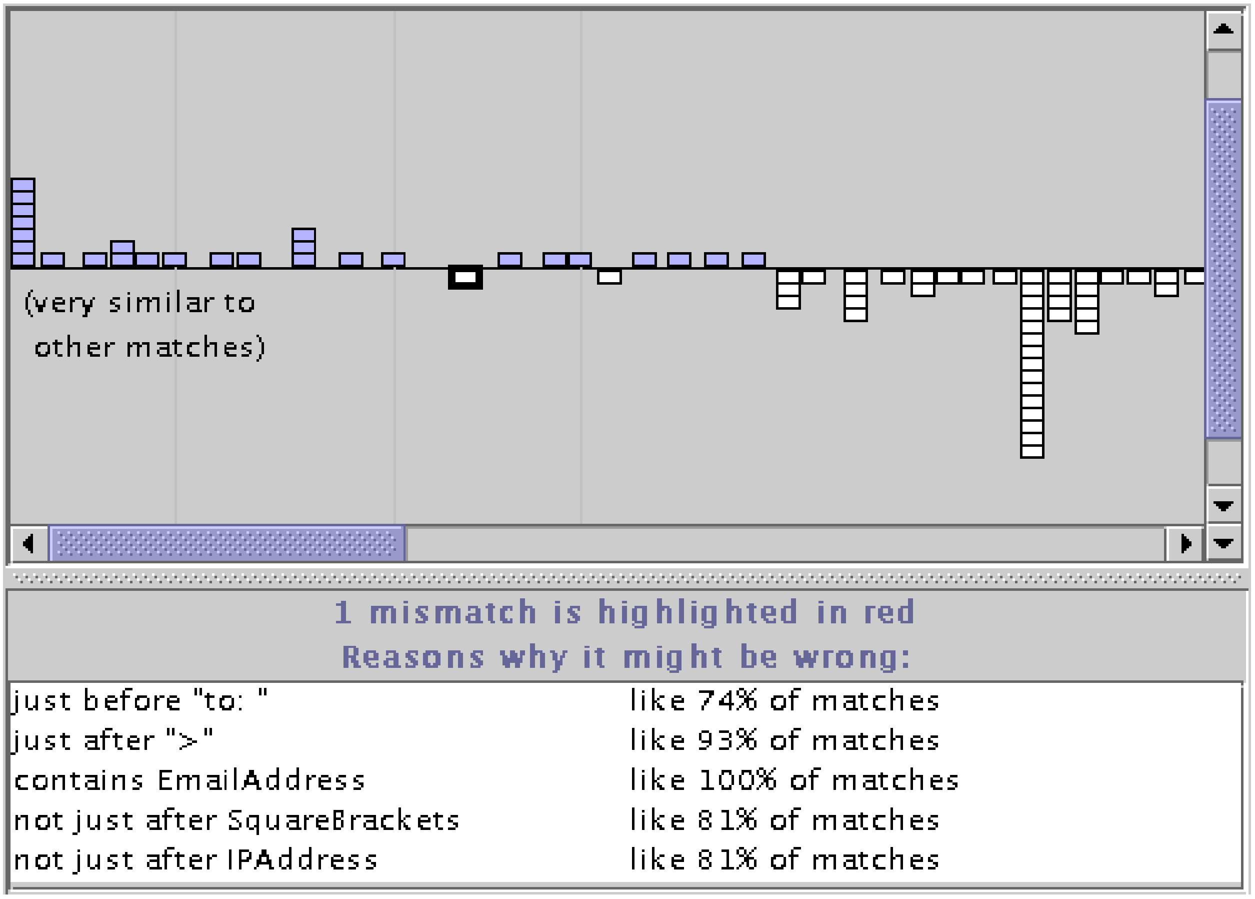
The tricky part of displaying mismatches is determining a set of candidate mismatches. The search space for pattern matching is the set of all substrings of the document. A naive approach would let the set of mismatches be the complement of the matches relative to this entire search space. Unfortunately, this set is quadratic in the length of the document. There are at least three reasonable ways to reduce the search space. Currently, LAPIS only implements the first:

1. **Negated constraint.** Many patterns are written by appending one or more constraints to a library pattern. For example, `Line containing "Truman"` constrains the *Line* pattern. If the user's pattern follows this scheme, then the constraint can be negated to find a set of candidate mismatches: `Line not containing "Truman"`. This technique effectively restricts the search space to the unconstrained library pattern, `Line`. The rules for finding this unconstrained pattern are the same as the rules used by the Keep command to find its record set (Section 8.1.3).

2. **All substrings between matches.** Since most applications of pattern matching (like find-and-replace) require nonoverlapping matches, a mismatch might be defined as any substring that does not overlap a match. Even though this set may still be quadratic, it can be represented compactly using region rectangles. This strategy has not been implemented in LAPIS.

3. **Approximate matches**. If the user specifies a literal string or regular expression pattern, then a set of mismatches can be generated by approximate string matching [WM92], which allows a bounded number of errors in the pattern match. This strategy has not been implemented either.

Regardless of how the possible mismatches are defined, the Unusual Matches window plots each mismatch on the same graph as the matches. Mismatches are colored white and plotted below the horizontal midline to clearly separate them from matches. Like matches, mismatches with identical feature vectors are clustered together into a stack. Clicking on a mismatch highlights it in the text editor and displays an explanation of why it should be considered as a possible match. The explanation consists of the highest-weighted features in which the mismatch agrees with the median match. Figure 10.5 shows the explanation for a mismatch.



1 mismatch is highlighted in red
Reasons why it might be wrong:

just before "to: "	like 74% of matches
just after ">"	like 93% of matches
contains EmailAddress	like 100% of matches
not just after SquareBrackets	like 81% of matches
not just after IPAddress	like 81% of matches

Figure 10.5: The Unusual Matches window showing both matches and mismatches to the pattern `Line starting "From:"` in a collection of email message headers. The most prominent mismatch, which is selected, is a Sender line which appears where the From line would normally appear in the message.

10.4 Implementation

This section describes the outlier finding algorithm. The algorithm takes as input a set of regions R and returns a ranking of R by each region's degree of similarity to the other members of R. Similarity is computed by representing each region in R by a binary-valued feature vector and computing the weighted Euclidean distance of each vector from the median vector of R. The distance calculation is weighted so that features which are more correlated with membership in R receive more weight.

Borderline mismatches are found by a related algorithm that takes two disjoint sets, R and $\overline{R}$, where R is the set of matches and $\overline{R}$ is the set of mismatches. The algorithm then ranks the elements of both sets according to their similarity to R. Since this algorithm is used to rank both matches and mismatches, it will be referred to below as the *two-sided outlier finder*, while the algorithm that ranks only the matches will be called the *one-sided outlier finder*. Outlier highlighting uses one-sided outlier finding. The Unusual Matches window uses the two-sided outlier finder when the user's pattern can be negated using the "negated predicate" technique described previously. Otherwise, the Unusual Matches window falls back to one-sided outlier finding. The discussion below focuses on one-sided outlier finding, mentioning the two-sided algorithm only where it differs.

The only part of these algorithms that is specific to text substrings is feature generation. Applying the algorithm to other domains would entail using a different set of features, but otherwise the algorithm would remain the same.

10.4.1 Feature Generation

The features used by the outlier finder take the same form as the features used by the selection guessing algorithm (Section 9.2.1). A feature is a TC expression of the form *`op F`*, where *`op`* is one of the TC operators `equals`, `just before`, `just after`, `starting`, `ending`, `in`, or `contains`, and *`F`* is either a literal or a pattern identifier.

Features based on pattern identifiers are generated by searching the document for every pattern in the pattern library and then applying the seven TC operators to each of the resulting region sets.

Literal features are generated by examining the text of the regions in R. The literal feature generation algorithms are similar to those used by selection guessing (Section 9.2.1), with one important difference. Selection guessing only needs features that describe all the positive examples. Outlier finding, on the other hand, needs features that match some, but not all, of the regions in R. Literal features that match only a few regions in R are not useful either. For example, `contains "apple"` is not a useful feature if only one region in R contains the string `apple`. Admitting a feature like this would require us to admit *all* substrings as features — e.g., `banana`, `pear`, `passionfruit` — which would greatly expand the feature list without gaining any particular leverage for finding outliers. Thus, it is necessary to set some threshold on the frequency of a literal feature. Based on the assumption that outliers are rare, the threshold used in LAPIS is simple majority. A literal feature must match at least half of the regions in R to be used for outlier finding.

At present, only four kinds of literal features are generated. (In the discussion below, the regions in R are denoted by $x_i[y_i]z_i$, for $1 \leq i \leq |R|$, in some arbitrary order. For any region i, the whole document is the string $x_i y_i z_i$ and y_i is the part of the document selected by region i.)

- `starting`: Find the longest common prefixes of every pair of y_i. This can be done efficiently by sorting the y_i and comparing every adjacent pair of strings. For each longest common prefix p, generate `starting "p"` if and only if p is a prefix of at least half the y_i.
- `ending`: find the pairwise longest common suffixes of the y_i that are suffixes of at least half the y_i, using an algorithm analogous to `starting`.
- `just before`: find the pairwise longest common prefixes of the z_i that are prefixes of at least half the z_i.
- `just after`: find the pairwise longest common suffixes of the x_i that are suffixes of at least half the x_i.

The remaining three operators, `equals`, `in`, and `contains`, are not used to generate literal features. Literal features using `equals` would be redundant with `starting` and `ending` features that are already generated. Literal features using `in` are either redundant with `starting` or `ending` or not useful. Literal features using `contains` are hard to generate efficiently. The suffix tree approach used by selection guessing only generates `contains` features that match all the regions in R, which would be useless for distinguishing outliers. Developing an efficient algorithm to find substrings that are common to at least half of R is an interesting problem for future work.

The two-sided outlier finder generates literal features by sorting both R and $\overline{R}$ together, so that it considers literal features shared by any pair of matches or mismatches. However, a literal feature must be shared by at least half of R or at least half of $\overline{R}$ to be retained as a feature.

10.4.2 Feature Weighting

After generating a list of features, the next step is determining how much weight to give each feature. Without weights, only the number of unusual features would matter in determining similarity. For example, without weights, two members of R that differ from the median in only one feature would be ranked the same by the outlier finder, even if one region was the sole dissenter in its feature and the other shared its value with 49% of the other members of R. We want to prefer features that are strongly skewed, such that most (but not all) members of R have the same value for the feature.

The one-sided outlier finder weights each feature by its inverse variance. Let $P(f|R)$ be the fraction of R for which feature f is true. Then the variance of f is

$$\sigma_f = P(f|R)(1 - P(f|R))$$

The weight for feature f is $w_f = 1/\sigma_f$ if $\sigma_f \neq 0$, or zero otherwise. With inverse variance weighting, features that have the same value for every member of R ($\sigma_f = 0$) receive zero weight, and hence play no role in the outlier ranking. Features that are evenly split receive low weight, and features that differ on only one member of R receive the highest weight ($|R|/(|R| - 1)$).

Two-sided outlier finding uses not only R but also $\overline{R}$ to estimate the relevance of a feature. We want to give a feature high weight if it has the same value on most members of R, but the opposite value on most of $\overline{R}$. To do this, the two-sided outlier finder uses *mutual information* to estimate the weights [Pap91]. The mutual information between a feature f and the partition $R, \overline{R}$ is given by

$$MI_f = H(R) - H(R|f)$$

where $H(R)$ is the entropy of R and $H(R|f)$ is the conditional entropy of R given f:

$$\begin{aligned} H(R) &= -P(R)\log P(R) - P(\overline{R})\log P(\overline{R}) \\ H(R|f) &= P(f)\left(-P(R|f)\log P(R|f) - P(\overline{R}|f)\log P(\overline{R}|f)\right) \\ &\quad +P(\overline{f})\left(-P(R|\overline{f})\log P(R|\overline{f}) - P(\overline{R}|\overline{f})\log P(\overline{R}|\overline{f})\right) \end{aligned}$$

Mutual information is related to the information gain heuristic used to induce decision trees [Qui86].

10.4.3 Feature Pruning

After computing weights for the features, the next step is pruning out redundant features. Two features are *redundant* if the features match the same subset of R (and $\overline{R}$) and one feature logically implies the other. For example, in a list of Yahoo URLs, the features `starting URL` and `starting "http://www.yahoo.com"` would be redundant. Keeping redundant features gives them too much weight, so the system keeps only the more specific feature and drops the other one.

Features are tested for redundancy by first sorting them by weight and comparing features that have identical weight. Since features are represented internally as region sets, the system can quickly compare the two region trees to test whether the matches to one feature are a subset of the matches to the other. Thus the system can find logical implications between features without heuristics or preprogrammed knowledge. It doesn't need to be told that `LowercaseLetters` implies `Letters`, or that `starting "http"` implies `starting URL`. The system discovers these relationships at runtime by comparing the region sets that these features match.

Pruning does not eliminate all the dependencies between features. For example, in a web page, `contains URL` and `contains Link` are usually strongly correlated, but neither feature logically implies the other, so neither would be pruned. The effect of correlated features could be reduced by using the covariances between features as part of the weighting scheme, but it is hard to estimate the covariances accurately without a large amount of of data. Accurately estimating the n^2 covariances among n features would require $O(n^2)$ samples. Another solution would be to carefully design the feature set so that all features are independent. This might work for some domains, at the cost of making the system much harder to extend. One of the benefits of the current approach is that new knowledge can be added to the outlier finder simply by writing a pattern and putting it in the library. Thus a user can personalize the outlier finder with knowledge like `CampusBuildings` or `ProductCodes` or `MyColleagues` without worrying about how the new patterns are related to existing patterns.

10.4.4 Ranking

The last step in outlier finding is determining a typical feature vector for R and computing the distance of every element of R from this typical vector.

To form the typical feature vector, the outlier finder computes the *median* value of each feature over all elements of R. Another possibility is the mean vector, but the mean of every nontrivial feature is a value between 0 and 1, so every member of R differs from the mean vector on most features and looks a bit like an outlier as a result. The median vector has the desirable property that when a majority of elements in R share the same feature vector, that vector is the median.

After computing the median feature vector $\overrightarrow{m}$, the outlier finder computes the weighted Euclidean distance $d(\overrightarrow{r})$ between every $\overrightarrow{r} \in R$ and $\overrightarrow{m}$:

$$d(\overrightarrow{r}) = \sqrt{\sum_f w_f (r_f - m_f)^2}$$

R is then sorted by distance $d(\overrightarrow{r})$. Elements of R with small $d(\overrightarrow{r})$ values are typical members of R; elements with large $d(\overrightarrow{r})$ values are outliers.

The two-sided outlier finder also computes $d(\overrightarrow{r})$ for members of $\overline{R}$. Members of $\overline{R}$ with small $d(\overrightarrow{r})$ values share many features in common with R, and hence are outliers for $\overline{R}$.

10.5 User Study

To evaluate the effectiveness of outlier highlighting, the simultaneous-editing user study from Section 9.3.1 was repeated with new subjects. The only difference between the original study and the new study was the presence of outlier highlighting. The new study's tutorial discussed what outlier highlighting looks like and what it means.

For the new study, six new users were found by soliciting campus job newsgroups (`cmu.misc.jobs` and `cmu.misc.market`). All were college undergraduates with substantial text-editing experience and varying levels of programming experience (3 described their programming experience as "little" or "none," and 3 as "some" or "lots"). Users were paid \$5 for participation.

10.5.1 Results

Comparing the two groups of users, one with outlier highlighting and the other without, showed a reduction in uncorrected inferences, although the sample size was too small for statistical significance. The system's incorrect inferences on the three tasks fall into four categories:

- Year (task 1): selection of the last two digits of the year (Figure 10.1)
- Author (task 1): selection of the author's name or the position just after it, which errs on "Hayes-Roth" (Figure 10.2)
- Winner (task 3): selection of the winning team's name or just after it, which errs on "Red Sox"
- Loser (task 3): selection of the losing team's name or just after it, which errs on (a different instance of) "Red Sox"

	Corrected selections	
Selection	No outlier highlighting	Outlier highlighting
Year	8/8 (100%)	7/7 (100%)
Author	0/7 (0%)	5/8 (63%)
Winner	1/8 (13%)	4/7 (57%)
Loser	4/7 (57%)	5/6 (83%)

Table 10.1: Fraction of incorrectly-inferred selections that were noticed and corrected by users (number corrected / number total). The denominators vary because some users never made the selection and others made it more than once.

	Tasks completed with errors	
Task	No outlier highlighting	Outlier highlighting
1	4/8 (50%)	2/6 (33%)
2	1/8 (13%)	2/6 (33%)
3	3/8 (38%)	1/6 (17%)

Table 10.2: Fraction of tasks completed with errors in final result (number of tasks in error / number total).

Only tasks 1 and 3 have incorrect inferences. All selections in task 2 are inferred correctly from one example. For all the incorrect inferences, outlier finding was perfect in the sense that every error was highlighted as an outlier, so overlooked errors are not due to the outlier finding algorithm. In these tasks, the outlier finder also highlighted regions that were not errors.

All users in both groups noticed that the Year selection was inferred incorrectly and corrected it, probably because the inference is dramatically wrong (Figure 10.1). For the other two kinds of selections, the outlier highlighting algorithm correctly highlighted the errors in the selection. As a result, users seeing the outlier highlighting corrected the Author, Winner, and Loser inferences more often than users without outlier highlighting (Table 10.1). In particular, the Author inference, which was *never* noticed or corrected without outlier highlighting, was noticed and corrected 5 out of 8 times (63%) with the help of outlier highlighting. Users confirmed the value of outlier finding by their comments during the study. One user was surprised that outlier highlighting was not only helpful but also conservative, highlighting only a few places.

Because outlier highlighting encouraged users to correct bad inferences, it also reduced the overall error rate on tasks 1 and 3, measured as the number of tasks finished with errors in the final result (Table 10.2). Note that editing with an incorrectly-inferred selection did not always lead to errors in the final output, because some users noticed the errors later and fixed them by hand.

Although outlier highlighting reduced the number of errors users made, it did not eliminate them entirely. One reason is that the system usually took 400-800 milliseconds to compute an inference, with or without outlier highlighting, and users did not always wait to see the inferred selection before issuing an editing command. For example, in the outlier-highlighting condition, 2 of the 3 uncorrected Author inferences went uncorrected because the user issued an editing command before the inferred selection and outlier highlighting even appeared. After the user study, outlier highlighting was changed so that records containing outliers remain highlighted in red through subsequent editing operations, until the user makes a new selection. As a result,

Task	Equivalent task size	
	No outlier highlighting	Outlier highlighting
1	8.4 recs [2.1–12.2 recs]	11.3 recs [8.1–18.0 recs]
2	3.6 recs [1.9–5.8 recs]	4.7 recs [3.2–7.2 recs]
3	4.0 recs [1.9–6.2 recs]	3.4 recs [2.5–4.0 recs]

Table 10.3: Equivalent task sizes (mean [min–max]) for each task in each condition.

even if the user doesn't notice an incorrect selection before editing with it, the persistent outlier highlighting hopefully draws attention to the error eventually.

Outlier highlighting also draws attention to correct inferences, undeservedly. Several users felt the need to deal with false outliers, to "get rid of the red" as one user put it. The design inadvertently encouraged this behavior by erasing the red highlight if the user provided the outlier as an additional example. As a result, several users habitually gave superfluous examples in order to erase all the outlier highlighting. Of the 143 total selections made by users with outlier highlighting, 16 were overspecified in this way, whereas no selections were overspecified by users without outlier highlighting. To put it another way, the tasks in the user study required an average of 1.26 examples per selection for perfect inference. Without outlier highlighting, users gave only 1.13 examples per selection, underspecifying some selections and making errors as a result. With outlier highlighting, users gave 1.40 examples per selection, overspecifying some selections. Giving unnecessary examples is not only slower but also error-prone, because the extra examples may be inconsistent with the desired selection. This happened to one user in task 2 — a correct inference became incorrect after the user misselected an outlier while trying to erase its highlight.

After the study, several design changes were made to mitigate the problem of overspecified selections. First, selecting an outlier as an additional example no longer erases its outlier highlighting. Instead, users who want to "get rid of the red" must right-click on an outlier to dismiss its highlight, eliminating the danger of misselection. This design was inspired by Microsoft Word, which uses the context menu in a similar fashion to ignore or correct spelling and grammar errors. Second, the outlier highlighting for simultaneous editing was changed so that only the record containing the outlier selection is colored red. The selections themselves remain blue, in order to make it clearer that the entire selection can be used for editing even if it contains outliers. There can be no ambiguity about which selection is the outlier, because each record in simultaneous editing contains exactly one selection.

Because outlier highlighting encourages users to attend to possible errors and give more examples, it also tends to slow down editing somewhat. Table 10.3 compares the equivalent task sizes with and without highlighting. The average equivalent task size with outlier highlighting was slightly larger for tasks 1 and 2, indicating that outlier highlighting made simultaneous editing slower on average. On task 1, extra editing time was well spent, because users corrected bad inferences. On task 2, all the inferences were correct after one example, so the extra editing time was wasted attending to false outliers. On task 3, however, the average equivalent task size actually decreased slightly. At least part of this decrease is explained by the fact that it takes less time to correct a selection error when it is noticed *before* editing with it (which happened more often in the outlier-highlighting condition) than it does to correct errors at the end of the task (which happened more often in the no-outlier-highlighting condition). A stitch in time saves nine.

Chapter 11

Conclusion

This concluding chapter discusses the main design decisions in lightweight structure, summarizes the contributions of the dissertation, and suggests some directions for future work.

11.1 Discussion

Lightweight structure offers a new way to describe and manipulate structure in text. A number of design decisions are essential to the idea of lightweight structure and its implementation in LAPIS. This section reviews these decisions and justifies them, in light of the supporting evidence presented in previous chapters.

The strongest evidence of the value of lightweight structure lies in the range of applications to which it has been applied: pattern matching (Chapter 6), multiple-selection editing (Chapter 7), Unix-style text processing and web automation (Chapter 8), selection guessing and simultaneous editing (Chapter 9), and outlier finding (Chapter 10) are all applications that benefit from lightweight structure. The user studies and examples in these chapters demonstrate the usefulness of these applications.

The applications also prove that one of the main goals of the thesis has been met: namely, that lightweight structure enables reuse of structure abstractions. Users can compose abstractions to write TC patterns and define new abstractions. Unix tools can exploit abstractions to filter, sort, and transform richly structured text, not just plain text files. Web scripts can employ abstractions to click on hyperlinks, fill in form fields, and extract data form the Web automatically. Machine learning agents can use abstractions in features to infer patterns from examples and detect potential pattern-matching errors. That the same library of structure abstractions could be reused in all these applications testifies to the generality of the lightweight structure idea.

One potentially controversial aspect of lightweight structure is the decision to discard the conventional syntax tree representation of text structure, in favor of region sets. Although syntax trees make an appearance in LAPIS, since both the HTML parser and Java parser generate trees in the course of parsing, the trees are only used to generate region sets and are then discarded. Region sets are the primary representation of structure used in LAPIS.

The region set representation has a number of advantages. It is easy to represent multiple independent hierarchies in the same document, such as the physical hierarchy (lines and pages), the logical hierarchy (sentences and paragraphs), or the syntactic hierarchy in source code (expressions and statements). Patterns and inferences can freely intermix references to abstractions from different hierarchies. No abstractions are preferred over any others by the region set representation, whereas a syntax tree representation tends to prefer patterns and operations on syntactic structure. The region set representation allows a lightweight structure system to support a range of approaches to text processing: lexical, syntactic, and hybrids.

One drawback of the region set representation, relative to a tree, is that some kinds of hierarchical queries are slower. In a tree, the children of a node can be found in $O(1)$ time simply by following pointers. With region sets representing the structure, however, finding the children of a region requires a query into a rectangle collection data structure, which typically takes $O(\log N)$ time. the applications to which lightweight structure is applied in this dissertation, however, finding the children of a *single* region is far less common than finding the children of a *set* of regions throughout a document, in which case the cost of traversing a rectangle collection can often be amortized across all the regions in the set using techniques like tandem traversal (Section 4.6.5) or plane-sweep intersection (Section 4.6.7).

When speed of access to one hierarchy is important, however, lightweight structure does not prohibit using a tree representation in concert with region sets. In fact, the rendered HTML view in LAPIS does precisely this. When a web page is displayed in LAPIS, it is represented both as a tree of HTML elements for efficient rendering, and as a string of content for matching and manipulation by region sets. Thus the region set representation can coexist with, and provide benefits to, a tree representation.

LAPIS also demonstrates that region-set structure can be added to an existing text editor or web browser without changing the system's internal data structures. LAPIS is designed around the `JEditorPane` component included in the Java library. Although the user interface of the component had to be overridden and extended substantially, mainly to permit rendering of multiple selections and editing with multiple cursors, the internal document representation used by `JEditorPane` was not modified at all. In fact, the components of LAPIS that implement lightweight structure — the region set data structures, algebra operators, and pattern language — are independent of the document representation. These components interact with the document through a generic interface, `Document`, which essentially presents a document as a string of content with a collection of metadata properties (Section 8.13). LAPIS actually includes several implementations of the `Document` interface. One is a wrapper for the internal document representation used by `JEditorPane`, which is used when LAPIS is running as a graphical user interface. Another implementation simply wraps around a `String`. This implementation is used when LAPIS is run from the command line without a graphical user interface (Section 8.14). Other editors or browsers could be supported by wrapping their internal representations with an implementation of `Document`. This level of independence is made possible by the fact that lightweight structure separates the representation of text from the structure that is used to manipulate it.

A key feature of the region algebra is the decision to support arbitrary region sets. Many systems support only flat structure, or only hierarchical structure, or only overlapping structure. The region algebra described in this dissertation can handle all of these kinds of structure, even combined into the same region set. The greatest benefit of this approach lies in its simplicity and uniformity. A user can write a pattern that refers to both `Line` and `Sentence` without paying undue attention to the fact that lines and sentences may overlap or nest in arbitrary ways. Although the cost of this uniformity is sometimes quadratic running time, the performance tests in Chapter 4 show that the common case is significantly better.

Supporting arbitrary region sets also opens the door to using the region set representation for *predicates* as well. This observation lies behind the decision to define unary operators in the region algebra. In other systems, a predicate like *contains PhoneNumber* can only be represented by an procedure, specifically one that tests whether a given region contains a phone number. In the region algebra, however, *contains PhoneNumber* can be represented explicitly by a region set — the set of all regions in the document that satisfy the predicate. A region set representing a predicate can be treated in the same way as a region set representing any other kind of structure. It can be searched, counted, combined with other region sets, and used for pattern matching, inference, or text manipulation. The explicit representation of predicates as region sets is essential to the interactive performance of selection guessing, simultaneous editing, and outlier finding, since predicate features can be precomputed, stored, and then used to form hypotheses or feature vectors.

Probably the most powerful advantage of lightweight structure — and the one most likely to lead to future innovations — is the way that all structure is treated as encapsulated abstractions with opaque implementations. From the user's point of view, `Sentence` is simply a set of sentence regions. The actual definition of `Sentence`, i.e., the parser that determines which regions are sentences and which aren't, is irrelevant. It might be a regular expression, a grammar, or a collection of TC patterns. It might also be a Bayesian classifier, a neural net, or some other learned classifier. It might be a collection of classifiers that vote, or an agent that accesses a dynamic structure library on the Web. It might even be a hybrid of machine intelligence and human intelligence, with a simple parser that correctly identifies 99% of the cases, leaving the remaining 1% (perhaps selected by outlier finding) to a human judge. The lightweight structure model opens up a wide spectrum of possibilities for structure description. Yet regardless of how structure is described, it can be used in all of the applications discussed in this dissertation: pattern matching, text processing, web automation, editing, inference, and outlier finding. More will be said at the end of this chapter about the larger possibilities of lightweight structure, for not only text but other kinds of media as well.

11.2 Open Questions

Like any thesis, lightweight structure raises new questions while answering others. This section gathers together some of the most salient and interesting questions about the specific ideas and techniques presented in this dissertation.

Region Set Model and Algebra (Chapter 3)

One limitation of the region set model is the difficulty of representing noncontiguous structure. Section 1.3 gave several examples of such structure, such as columns in plain text or HTML tables. One way to get around this limitation might be to use *sets* of region sets — i.e., representing each column as a region set, and the set of columns as a set of region sets. This extension should be taken with some care, however, to avoid combinatorial explosion; although the maximum size of a region set is only $O(n^2)$, a set of region sets may be $O(2^n)$.

Another kind of noncontiguous structure is linking. Hyperlinks, function calls, variable references, and bibliographic citations are all examples of linking structure. Links may represent relations not just within documents, which is the kind of structure that has occupied the interest in this thesis, but *between* documents as well. Representing linking in the region set model would call for two changes. First, each region would have to be represented as a triple (document, start offset, end offset), and region set data structures would have to be extended to store regions from multiple documents in the same set. Second, the links themselves might be represented by document-generated relations mapping the region at one end of the link to the region at the other end, and vice versa. Rectangle collection data structures might be useful for storing these relations, once they have been parsed from a set of documents.

Representing two-dimensional relationships in page layout is another interesting issue. The region set model described in this thesis treats every document as a one-dimensional string, so the six fundamental relations *before*, *after*, *overlaps-start*, *overlaps-end*, *in*, and *contains* are sufficient basis for the region algebra. When a document is rendered on screen or on paper, however, it becomes two-dimensional, and its components may have two-dimensional relationships, such as *above*, *below*, *left*, and *right*. The region algebra could either be extended to capture these relationships, or they could be represented like linking structure, as document-specific relations generated by the renderer.

Region Algebra Implementation (Chapter 4)

LAPIS implements only two rectangle collection data structures, RB-trees and region arrays. It would be instructive to implement some of the other alternatives described in Chapter 4, such as quadtrees, 4D point collections, and plane-sweep intersection, and compare performance.

The measurements at the end of Chapter 4 suggest that TC pattern matching might outperform regular expression matching in some cases. A head-to-head comparison would require using the fastest regular expression package available, however, which would certainly be written in C or C++, so the region algebra implementation would also have to be rewritten in a higher-performance language to make the comparison fair.

Scaling up to large document collections is another important area of future work. Applying all the patterns and parsers in the LAPIS library, which is necessary for inference and outlier finding, takes about a minute per megabyte of text scanned (Table 4.6). Alternative data structures or high-performance implementation languages may reduce this time, but only by a constant factor, and the library is very likely to grow as the user adds more patterns and parsers. Long-term solutions may involve smarter decisions about which patterns or parsers to try (e.g., only run Java-related parsers and patterns on files that end in `.java`), or caching region sets on disk to avoid repeated parsing. Fortunately, the RB-tree used in LAPIS is derived from a disk-based data structure, so working with on-disk RB-trees would be easier.

Language Theory (Chapter 5)

Several questions remain to be answered about the recognition power of the region algebra, mostly involving the effect of the *forall* operator. It seems clear that $RSTA_\emptyset$ is more powerful than $RST_\emptyset$, but characterizing the class of languages recognized by $RSTA_\emptyset$ is still an open question, as are the languages recognized by $RSTA_\mathcal{F}$ and $RSTA_\mathcal{CFL}$. It would also be interesting to extend the region algebra with operators from other pattern languages — for example, the Kleene star from regular expressions, or recursive productions from context-free grammars — and study the effect on the class of languages that can be recognized.

TC Pattern Language (Chapter 6)

Probably the most valuable addition to the TC pattern language would be a facility for defining generic, parameterized structure abstractions. The HTML parser already uses this idea to a limited extent by embedding "parameters" in its abstraction names: a tag name in angle brackets (e.g., `<img>`) matches a start tag, and a tag name in square brackets (`[body]`) is an element. These abstractions are not truly parameterized, however, since the parser only recognizes a finite set of them, one for each tag defined in HTML 4.0. An XML parser using this kind of naming scheme would have to support arbitrary tag names, a good reason to introduce parameterization.

User-defined relational operators would be another way to introduce parameterization into TC. For example, program code might be searched for `Method` ***named*** `"copy"`, or an email archive for `Message` ***from*** `"John"`, where the operators ***named*** and ***from*** are user-defined relations. Forming database-like queries would be easier if the language supported numeric and date relations, such as `Price < 100`. The most generic way to support these kinds of constraints would be to embed script code in a pattern that tests each region in a region set to see if it satisfies the predicate.

Another useful way to compose parsers is to use the output of one parser to determine the input to another. For example, Java comments may include HTML tags, so it makes sense to apply the Java parser first to find the comments, then apply the HTML parser just to the comments. The rendered view in LAPIS is another example (Section 6.2.11), since it is generated by the HTML parser by stripping out tags and then used as input for other parsers and patterns. Specifying these kinds of relationships between parsers would require a parser-control language, which might use TC expressions to specify how data should flow between parsers.

LAPIS (Chapter 7)

The highlighting techniques used in LAPIS are not well-suited to displaying nested or overlapping region sets. Several proposals for improvement are shown in Figure 7.5.

Multiple-selection editing likewise only supports editing with flat region sets. In hierarchical structure, like source code or XML, it may sometimes be useful to apply editing commands to a nested region set. Extending the semantics of familiar editing commands to nested selections, and testing whether users understand them, would be interesting.

LAPIS can currently only edit plain text, not rendered HTML. Since the `JEditorPane` editor component on which LAPIS is based is capable of editing rendered text, extending LAPIS to do likewise would not be terribly difficult. More challenging, but more rewarding, would be integrating some or all of the features of LAPIS into an existing application like Emacs or Mozilla. The hardest part of the integration would probably be overriding the single-selection behavior that is deeply ingrained in most applications. If this obstacle can be overcome, however, the benefits would be significant, since both Emacs and Mozilla have large user bases who could provide more experience with the usability of these features in daily use.

Large documents with many selections pose a problem for multiple-selection editing. Making a million edits with every keystroke may slow the system down to a crawl, particularly if the text editor uses a *gap buffer* to store the text [Fin80]. Gap buffers are used by many editors, including Emacs and `JEditorPane`. With a gap buffer and a multiple selection that spans the entire file, typing a single character forces the editor to move nearly every byte in the buffer. One way to address this problem is to delay edits to distant parts of the document until the user scrolls to them or saves the document. Another solution might be to have multiple gaps in the buffer, one for each selection.

Unix Tools and Browser Shell (Chapter 8)

The Unix tool set includes many more tools that might be usefully adapted to lightweight structure. Some likely candidates are:

- `diff`, for comparing two region sets in the same or different documents;
- `uniq`, for eliminating duplicates in a region set, where the notion of a duplicate may be generalized beyond simple character-by-character identity;
- `patch`, for merging differences between two region sets.

One drawback of the LAPIS tools, compared to their Unix counterparts, is that processed files are loaded entirely into memory. The LAPIS tools would scale better if they operated with a single pass across a file, using bounded working memory. Single-pass operation may not always be possible, since TC patterns can depend on nonlocal context which may require access to the entire file, but it would be desirable whenever possible.

Several features are needed to make the browser shell more useful and more efficient as a command shell, including:

- **Background processes.** Web browsers generally stop loading a page when a new URL is typed in the Location box. Similarly, LAPIS automatically stops the currently executing command when a new command is typed in the Command box. As a result, only one command can be running in each LAPIS browser window. An improvement would be support for background-process syntax. If a command ends with &, it could continue running in the background, saving its output in case the user ever backs up through the history.
- **Handling large outputs.** A command may generate too much output for the browser to display efficiently. The same problem often happens in typescript shells, usually forcing the user to abort the program and run it again redirected to a file. To handle this problem, the browser could automatically truncate the display if the output exceeds a certain user-configurable length. The remaining output would still be spooled to the browser cache, so that the entire output can viewed in full if desired, or passed as input to another program.
- **Streaming I/O.** A pipeline may process too much data for the browser's limited cache to store efficiently. Although the browser shell's automatic I/O redirection could still be used to assemble the pipeline (presumably on a subset of the data), the pipeline would run better on the real data if its constituent commands were invoked in parallel with minimal buffering of intermediate results. The browser shell might do this automatically when invoking a script, since the intermediate results of a script do not need to be shown unless it is being debugged.

- **Shell syntax.** Expert users would be more comfortable in the browser shell if it also supported conventional operators for pipelining and I/O redirection, such as |, <, >, and > >. The most direct way to accommodate expert users might be to embed an existing shell, such as `bash` or `tcsh`, as an alternative to Tcl.

Inference (Chapter 9)

The inference algorithms implemented in LAPIS only infer multiple selections in a single file. It would be straightforward to extend them to infer selections across multiple files, such as a collection of web pages or source code modules, but some additional issues would arise.

First, when inference is applied to multiple files, or large single files, it becomes harder for the user to check for incorrect inferences. Outlier highlighting helps, but it requires the user to browse through the documents to look at the outliers. One visualization that might avoid scrolling is a "bird's-eye view" showing the entire file, with greeked text, so that deviations or outliers in an otherwise regular selection can be noticed at a glance. Another useful visualization would be an abbreviated context view, showing only the lines containing selections.

Another question that must be answered for editing multiple files by inference is where the data should reside. For simultaneous editing at interactive speeds, all the documents to be edited must fit in main memory, with some additional overhead for parsing and storing feature lists. For large data sets, this may be impractical. However, it is easy to imagine interactively editing a small sample of the data to record a script which is applied in batch mode to the rest of the data. The batch mode could minimize its memory requirements by reading and processing one record at a time (or one translation unit at a time, if it depends on a Java or HTML parser). Scripts recorded from simultaneous editing would most likely be more reliable than keyboard macros recorded from single-cursor editing, since simultaneous editing finds general patterns representing each selection. The larger and more representative the sample used to demonstrate the script, the more correct the patterns would be.

The script could also be saved for later reuse, which raises another question. Simultaneous editing is designed for repetitive tasks where all the instances of the task are available at the same time. This class of tasks might be described as *repetition over space*. Another important class of tasks are those in which instances arise one by one over a period of time, which might be termed *repetition over time*. Email filtering is a good example. The inference techniques described in this thesis might be applied to this kind of task by first converting it to a repetition over space by collecting a set of instances over a period of time. Simultaneous editing could then be applied to those instances, recording a script that for future instances of the task. At some point in the future, the recorded script may encounter an instance that it cannot handle correctly, so the system needs some way to detect these cases, perhaps using outlier finding, and let the user fix the script with more simultaneous editing.

The inference techniques in simultaneous editing could be improved in several ways. The one-selection-per-record bias is a powerful heuristic, but it is not always sufficient. Allowing the user to override the bias, by making multiple selections in some records or removing all selections in other records, would allow the user to do nested iterations (e.g., over the varying numbers of parameters in each method declaration) or conditionals (e.g., editing public methods one way, private methods a different way) without leaving simultaneous editing mode. Currently, simultaneous editing can handle these nested iterations and conditionals only by choosing different record sets, which is not particularly natural. Inference can also be improved by specifying different weights for different library abstractions. For example, when the user is editing Java code, features involving Java syntax might be preferred by the inference algorithm over lexical features like capitalization or whitespace.

Since the user studies described in Chapter 9 tested selection guessing and simultaneous editing in isolation, it is still an open question whether users can understand the difference between the two modes and determine when to use each one, or whether the two modes should be somehow combined into one.

Finally, it would be interesting to implement and evaluate simultaneous editing in other kinds of applications. For example, spreadsheets often have sequences of similar formulas which might be edited simultaneously. A column in a relational database or a set of related filenames (e.g., `*.cpp`) in a file manager might be edited in the same way.

Outlier Finding (Chapter 10)

One problem with outlier highlighting is that it requires the user to scan the document to look at the outliers, which does not scale well. In addition to highlighting outliers, it may also make sense for the system to save up outliers for later proofreading and allow editing commands to be undone on individual records.

Outlier finding suggests a new way to do global find-and-replace. Traditional find-and-replace commands offer two options: replacing matches one at a time with confirmation, or replacing all matches at once. Outlier finding offers a third way: replacing outlier matches one at a time with confirmation, but replacing typical matches all at once. Designing a user interface that seamlessly integrates outlier finding into the find-and-replace task would be an interesting future work.

The Unusual Matches dialog is a step in the direction of a general pattern debugger, which might be as useful for awk and Perl programmers as it is for LAPIS users. Possible improvements include better explanations, the ability to fix a pattern using features from the explanation, and the ability to indicate that a feature is irrelevant. The outlier finding algorithm could also be improved by supporting literal `contains` features and scalar and real-valued features.

Outlier finding can be applied to testing and debugging in more general programming domains as well. An outlier finder might highlight unusual variable values, unusual procedure calls, and unusual event sequences. New visualizations would be needed to highlight these kinds of objects. Other research issues include developing a set of features for judging whether values and events are unusual, and exploring the tradeoff between precision and recall (i.e., highlighting too little vs. highlighting too much).

More Applications

Lightweight structure has a number of text-processing applications beyond the ones discussed in this dissertation.

Lightweight structure would be helpful for defining structure detectors for Apple Data Detectors [NMW98] and Microsoft Smart Tags [Mic02]. In particular, the ability to describe patterns by giving examples would make it easier for non-programming users to define their own data detectors or smart tags.

Copy-and-paste is another feature that could be improved by lightweight structure. Applications like bibliography editors, calendars, and contact managers represent data objects in structured fashion, usually presenting a form interface to the user for editing. Yet new citations, appointments, and contact information often arrive as unstructured or semi-structured text in email messages or web pages. Lightweight structure would enable the construction of *clipboard transducers* that convert between plain text format and the structured formats expected by other applications. Ideally, a user would be able to copy a citation from a web page or a signature line from an email message, and paste directly into a bibliography manager or contact manager, with the transducer automatically taking care of parsing it into structure and filling in the appropriate fields in the other application's interface.

Another potential application for lightweight structure lies in adapting web pages to resource constraints, such as low bandwidth and small screens. The user might describe the format of a web site by writing patterns or inferring them from examples, and then specify how pages in that format should be filtered or transformed when the site is viewed on a device with limited resources, such as a personal digital assistant or a cellphone.

11.3 Summary of Contributions

This dissertation introduces the idea of lightweight structure as a way to describe structure in text. Lightweight structure consists of the region set model, an extensible library of structure abstractions, and an algebra for combining region sets.

Although the region algebra is simple, consisting of six fundamental relational operators and the set operators, Chapter 3 shows that many other pattern-matching operators can be derived from it. Furthermore, when regions are interpreted geometrically as points in region space, the fundamental region relations correspond neatly to rectangles, which motivates the efficient rectangle-collection implementation described in Chapter 4.

The languages recognized by region algebra expressions are defined formally in Chapter 5, and found to depend on the recognition power of the abstractions referenced by the expression. Region algebra expressions using only literal string matching (and omitting *forall*) recognize the same class of languages as generalized star-free regular expressions. Region algebra expressions over regular abstractions (i.e., regular expressions in addition to literal strings) recognize regular languages, but algebra expressions over context-free expressions recognize a strict superset of the context-free languages.

The reusability of lightweight structure has been demonstrated by employing it in a variety of applications. The most natural application, of course, is pattern matching. The TC pattern language described in Chapter 6 is based very closely on the region algebra, so that TC patterns can compose and reuse structure abstractions. TC is also novel for the way it uses indentation to structure infix expressions and avoids Boolean operators where possible. A user study showed that users can generate and comprehend TC patterns, and the other applications of lightweight structure rely heavily on TC patterns to give feedback to the user.

LAPIS is another application of lightweight structure, showing how lightweight structure can be incorporated into a web browser or text editor. In LAPIS, the selection is a region set. Selections can be made by the mouse, by selecting patterns from the library of structure abstractions (including HTML and Java syntax), by writing a TC pattern, or by some combination of these techniques. Displaying a region set in a document requires some new highlighting techniques, which are discussed in Chapter 7. That chapter and the following one also show how a region set selection can be used for editing and text processing, generalizing some common Unix tools like `grep` and `sort` to the world of lightweight structure.

LAPIS also incorporates a scripting language, based on Tcl, designed for text processing with lightweight structure. Chapter 8 describes the language and showed how it can be used for web automation. LAPIS integrates the script interpreter with web browsing, text processing, and external command invocation to produce a novel user environment, the browser shell.

Lightweight structure has also been applied to machine learning. Chapter 9 shows two techniques for inferring region set selections from examples given by the user. Selection guessing, the more general technique, infers TC patterns using any of the structure abstractions in the library as features. Simultaneous editing, the second technique, adds a heuristic — restricting the inference to hypotheses that make exactly one selection in every record being edited — that reduces the hypothesis search space so much that most selections can be made with only one example. Both techniques use multiple forms of feedback to keep the user informed, displaying the inference both as a multiple selection and as a pattern. User studies showed that even users untrained in lightweight structure and TC pattern matching can use selection guessing and simultaneous editing for repetitive editing tasks.

Finally, this dissertation introduces the idea of outlier finding as a technique for reducing errors in pattern matching or repetitive text editing. Outlier finding is used in two places in LAPIS: to highlight unusual selections during simultaneous editing, and to generate the Unusual Matches dialog. Lightweight structure plays a role in outlier finding as well, since structure abstractions are used as features.

11.4 Looking Ahead

Lightweight structure raises the level of abstraction at which users interact with data. Instead of being limited to characters, words, or lines, like most text-processing systems, lightweight structure allows the user and the system to communicate at a higher level, about HTML elements, Java expressions, English sentences, or any other abstractions that might be installed in the library. As this dissertation has shown, raising the abstraction level brings improvements to pattern matching, Unix-style text processing, web automation, repetitive editing, inference of patterns from examples, and error detection.

There are two natural directions one could proceed — two dimensions along which a future research plan might be mapped out. One way leads deeper into the rich structure of text, or more accurately, to higher levels of abstraction. The other way leads to greater breadth, expanding the idea of lightweight structure beyond text into other domains.

First consider depth. This dissertation concerned itself chiefly with lexical and syntactic structure, but text is also rich with *semantic* structure. For example, a large body of research has been devoted to the problem of automatic text summarization, resulting in algorithms that can automatically reduce a long document to a short summary by locating and extracting pivotal sentences. If these algorithms were integrated into a LAPIS-like system and placed under user control, then a user could highlight individual threads of discussion in a document or collection of documents, focus on salient sentences, and quickly gain a sense of the meaning of the document, answer a question, or generate an abstract. Such a tool would be a boon for digesting, analyzing, and report-writing, problems which are increasingly important but not as yet supported by automated tools.

Source code also abounds with semantic information, including types, inheritance relationships, data flow, and control flow. Modern optimizing compilers incorporate a variety of advanced static analyses, such as interprocedural flow analysis and pointer alias analysis, which are largely hidden from the programmer's eye. If these analyses were exposed in a LAPIS-like interface, so that the user could view, constrain, and correct them in critical areas of a program, then the combination of compiler and human might be able to understand the code much better, helping the compiler to make it faster and the human to maintain it.

Style, in both writing and programming, is another interesting kind of semantic structure. Tools that evaluate style and help writers and programmers improve their style would be a fertile field of future work. Primitive style advice can be found in grammar checkers, `lint`, and compiler warnings, but these systems are ad-hoc, limited, and inextensible. The lightweight structure approach suggests that style knowledge could be built up like lexical and syntactic knowledge, as a library of reusable concepts that can be shared, adapted to the needs of individuals or organizations, and evolved over time.

Looking beyond text, structure can be found and exploited in many other kinds of data, including sound, 2D graphics, 3D graphics, and video. Like text, structure in other domains may exist on several different levels, some of which are more accessible to automatic detection than others. For example, in images, low-level structure like edges, corners, and areas of color are easy to detect; high-level structure like automobiles and buildings are much harder to identify. Nevertheless, a uniform framework for describing and composing structure detectors may have as much value in other domains as it does in text.

Take video as an example. As video begins to rival the written word as a communications medium, even ordinary users will need to browse, search and edit digital video. Algorithms developed for computer vision, such as object tracking and face recognition, could help with many of these tasks, but only if the algorithms are integrated into a usable human interface in which the user can control, constrain, and correct their results. Many of the techniques in LAPIS, such as the usable pattern language, simultaneous editing, and outlier finding, might have useful analogs in video editing.

Unlike text, other kinds of media draw a sharp distinction between structured and unstructured data. For example, there are two kinds of 2D graphical editors: drawing editors, which represent an image as a structured collection of graphical objects like rectangles, circles, and lines; and paint programs, which represent an image as an unstructured array of pixels. Likewise, in digital audio, one can use a MIDI editor to edit structured music files, or a waveform editor to mix unstructured digital audio. Lightweight structure suggests a third alternative: editing unstructured data as if it were structured, which might be a powerful way to unify the structured and unstructured application models.

Text will continue to be important, and lightweight structure can help. The techniques developed in this dissertation form a solid foundation for future tools, helping users bring the power of abstraction and automation to bear on their text processing problems.

Appendix A

Pattern Library

This appendix lists the pattern identifiers built into the LAPIS library, describing what each identifier is designed to match. When an identifier is defined by TC patterns, the TC patterns are specified afterward.

A.1 Business

Identifiers in the `Business` namespace are defined by TC patterns in the file `USEnglish.tc`.

Address

`State`	matches U.S. state names and their two-letter abbreviations.
`ZipCode`	matches 5-digit and 9-digit U.S. zip codes appearing just after a `State` in an address.

```
State is either "Alabama" or "Alaska"
                ...
             or "Wisconsin" or "Wyoming"
             or word = case-sensitive
                           either "AL" or "AK"
                                  ...
                               or "WI" or "WY"
                       ignoring nothing

@ZipCode is Number equal to /\d\d\d\d\d/
                 ignoring nothing
              just after State
ZipCode is flatten either @ZipCode
                   or @ZipCode
                       then "-"
                       then Digits equal to /\d\d\d\d/
                       ignoring nothing
```

Date

`DayOfMonth` matches a numeric day of the month, 1-31, appearing in a date, with optional “th”, “nd”, or “st” suffix.

`DayOfWeek` matches the English name of a weekday or its three-letter abbreviation.

`LongMonth` matches the English name of a month or its three-letter abbreviation.

`ShortMonth` matches a numeric month, 1-12, appearing in a date.

`Month` matches either `LongMonth` or `ShortMonth`

`LongYear` matches a four-digit year from either the 20th or 21st centuries

`ShortYear` matches a two-digit year appearing in a date.

`Year` matches either `LongYear` or `ShortYear`

`Date` matches a date, which contains at least a month and a year, and optionally a day of the month, day of the week, and a time.

```
@DayOfMonth is Number equal to /[12][0-9]|3[01]|0?[1-9]/
                     ignoring nothing
@DayOfMonth is either @DayOfMonth
                  or @DayOfMonth
                       then "th"
                       ignoring nothing
                  or @DayOfMonth
                       then "nd"
                       ignoring nothing
                  or @DayOfMonth
                       then "st"
                       ignoring nothing

@DayOfWeek is Word equal to either "Sun"
                               or "Sunday"
                               or "Mon"
                               or "Monday"
                               or "Tue"
                               or "Tues"
                               or "Tuesday"
                               or "Wed"
                               or "Wednesday"
                               or "Thu"
                               or "Thurs"
                               or "Thursday"
                               or "Fri"
                               or "Friday"
                               or "Sat"
                               or "Saturday"
                   ignoring nothing

@LongMonth is Word equal to either "Jan"
                               or "January"
                               or "Feb"
                               or "February"
                               ...
                               or "Dec"
                               or "December"
                        ignoring nothing

@ShortMonth is Number equal to /1[012]|0?[1-9]/
                        ignoring nothing

@Month is either LongMonth or @ShortMonth

@LongYear is Number equal to /(19|20)\d\d/
                        ignoring nothing

@ShortYear is Number equal to /\d\d/
                         ignoring nothing

@Year is either LongYear or @ShortYear

Date is flatten either
                       either @LongMonth
                               then @DayOfMonth
                               then @LongYear
                       or @LongYear
                           then @LongMonth
                           then @DayOfMonth
                       or @DayOfMonth
                           then @LongMonth
                           then @LongYear
                       ignoring either Spaces
                                    or Punctuation

                or either @LongMonth then @DayOfMonth
                       or @LongMonth then @LongYear
                       ignoring either Spaces
                                    or Punctuation
```

```
                    or either @Month
                               then @DayOfMonth
                               then @Year
                           or @Year
                               then @Month
                               then @DayOfMonth
                           or @DayOfMonth
                               then @Month
                               then @Year
                          ignoring /[-\/]/

                   or @DayOfWeek
                         then @LongMonth
                         then @DayOfMonth
                         then Time
                         then Word
                         then @LongYear
                         ignoring Spaces

    DayOfMonth is @DayOfMonth in Date

    DayOfWeek is @DayOfWeek

    LongMonth is @LongMonth

    ShortMonth is @ShortMonth in Date
                              not in Time
    Month is either LongMonth or ShortMonth

    LongYear is @LongYear

    ShortYear is ShortYear in Date
                           not in Time

    Year is either LongYear or ShortYear
```

Money

`Money`	matches a number preceded by a dollar sign

```
    Money is "$" then Number
                 ignoring nothing
```

Number

`Number` matches a number with optional comma separators and decimal point. Comma separators are recognized only up to 999,999,999. A better definition of `Number` would use a regular expression, as `ScientificNotation` does.

`ScientificNotation` matches a number with optional exponent (E+/-).

```
@Number is either Digits
              or Digits
                   then ","
                   then Digits
                   ignoring nothing
              or Digits
                   then ","
                   then Digits
                   then ","
                   then Digits
                   ignoring nothing
@Number is either @Number
               or @Number
                    then "." then Digits
                    ignoring nothing
Number is either @Number
              or "-" not just after Word
                  then @Number
                  ignoring nothing
ScientificNotation is /-?\d+(\.\d+)?(\s*[Ee][-+]?\d+)?/
```

PhoneNumber

`PhoneNumber` matches a 7-digit or 10-digit US phone number separated by dashes.

`AreaCode` matches the 3-digit area code that starts a 10-digit phone number.

`FaxNumber` matches a phone number labeled by the word “fax” before or after it.

```
@PhoneNumber is Number equal to /\d\d\d/
                then "-"
                then Number equal to /\d\d\d\d/
                ignoring nothing
AreaCode is Number equal to /\d\d\d/
                   ignoring nothing
                   just before @PhoneNumber
```

```
PhoneNumber is either @PhoneNumber
                   or AreaCode then "-"
                               then @PhoneNumber
                   or AreaCode then @PhoneNumber
                   or "(" then AreaCode
                          then ")" then @PhoneNumber
FaxNumber is PhoneNumber
                   either just after "fax"
                       or just before "fax"
                   ignoring either Punctuation
                                or Spaces
```

Time

`Time` matches a 12-hour or 24-hour time specification, with optional seconds and optional AM or PM.

```
Time is Number equal to /[012]?\d/
         then ":"
         then Number equal to /\d\d/
         ignoring nothing
Time is either Time
            or Time
                  then ":"
                  then Number equal to /\d\d/
                  ignoring nothing
Time is either Time
            or Time then Word equal to either "am"
                                           or "pm"
                              ignoring nothing
Time is either Time
            or "midnight"
            or "noon"
```

A.2 Characters

Identifiers in the `Characters` namespace are defined by the Java class `lapis.parsers.CharacterParser`.

`Token` matches runs of non-whitespace characters.

`Alphanumeric` matches runs of alphanumeric characters (letters or digits).

`Digits` matches runs of digits.

`Letters` matches runs of letters.

`LowerCaseLetters` matches runs of lowercase letters.

`UpperCaseLetters` matches runs of uppercase letters.

`Punctuation` matches runs of punctuation.

`Whitespace` matches runs of whitespace characters.

`Linebreak` matches individual linebreak characters.

`Spaces` matches runs of space characters.

`Tab` matches individual tab characters.

A.3 English

Identifiers in the `English` namespace are defined by TC patterns in the file `USEnglish.tc`.

Sentence

`Sentence` matches English “sentences” that start with a capitalized word and end with a period, exclamation point, or question mark.

```
@EndingPunctuation
   is either "." not just after case-sensitive
                                 either "Dr"
                                     or "Mr"
                                     or "Mrs"
             or "?"
             or "!"
          either just before Whitespace
                                 then UppercaseLetters
              or ending Paragraph
@StartingWord
   is CapitalizedWord
         either just after @EndingPunctuation
             or starting Paragraph

Sentence is from @StartingWord
            to @EndingPunctuation
              in Paragraph
              contains Spaces
```

Word

`Word` matches runs of alphanumeric characters.

`AllCapsWord` matches a word which consists entirely of uppercase letters.

`CapitalizedWord` matches a word starting with uppercase letters.

`LowerCaseWord` matches a word consisting entirely of lowercase letters.

`MixedCaseWord` matches a word with both uppercase and lowercase letters, where at least one uppercase letter occurs strictly inside the word.

```
Word is Alphanumeric
CapitalizedWord is Word starting UppercaseLetters
                        ignoring nothing
AllCapsWord is Word equal to UppercaseLetters
                    ignoring nothing
LowerCaseWord is Word equal to LowercaseLetters
                      ignoring nothing
MixedCaseWord is Word
                  contains LowercaseLetters
                           then UppercaseLetters
                           ignoring nothing
```

A.4 HTML

Identifiers in the `HTML` namespace are defined by the Java class `lapis.parser.HTMLParser`.

`Attribute` matches a name-value attribute in an HTML tag.

`name``-attr` matches an attribute named *name*. For example, `href-attr` matches the `href` attribute in an `<a>` tag. An identifier of this form is defined for every attribute named in the HTML 4.0 specification.

`AttributeName` matches the name part of an attribute (the part before the equals sign, or the entire attribute if no value is given).

`AttributeValue` matches the value part of an attribute (the part after the equals sign).

`Element` matches a complete element, running from its start tag to its matching end tag (if any).

`[`*`tagname`*`]` matches a complete element named *tagname*. For example, `[body]` matches the part of the document running from `<body>` to `</body>`. An identifier of this form is defined for every tag in HTML 4.0.

`Tag` matches any tag, which may be either a start tag or an end tag.

`StartTag` matches any start tag.

`<`*`tagname`*`>` matches a start tag named *tagname*. An identifier of this form is defined for every tag in HTML 4.0.

`EndTag` matches any end tag.

`</`*`tagname`*`>` matches an end tag named *tagname*. An identifier of this form is defined for every tag in HTML 4.0.

`Text` matches a run of text between tags.

A.5 Internet

Identifiers in the `Internet` namespace are defined by TC patterns found in the file `Internet.tc`.

`EmailAddress` matches an email address in conventional `user@hostname` form.

`Hostname` matches either an IP address or a domain name.

`IPAddress` matches an IP address in n.n.n.n form.

`RootDomain` matches the most common root domain names, such as `edu`, `com`, and `org`.

`URL` matches a Uniform Resource Locator.

```
EmailAddress is Token containing "@"
                  trim off Punctuation
IPAddress is Digits then "."
                    then Digits
                    then "."
                    then Digits
                    then "."
                    then Digits
                     ignoring nothing

RootDomain is Word equal to either "com"
                                or "edu"
                                or "gov"
                                or "mil"
                                or "org"
                                ...
                                or "it"
                                or "fi"
                   just after "."
                         ignoring nothing
```

```
    Hostname is either IPAddress
                    or /[\w\-\.]+/
                         ending "." then RootDomain
     URL is from case-sensitive either "http:"
                                    or "ftp:"
                                    or "mailto:"
                                    or "file:"
                                    or "gopher:"
                                    or "news:"
                                    or "nntp:"
                                    or "https:"
                                    or "telnet:"
                                    or "wais:"
                                    or "prospero:"
                                    or "javascript:"
            to point just before either Whitespace
                                     or ’"’
                                     or "’"
                                     or ">"
```

A.6 Java

Some of the identifiers in the `Java` namespace are defined by the class `lapis.parsers.JavaParser`. Others are defined by TC patterns in the file `Java.tc`.

`ActualParameter` matches an expression passed as a parameter to a method call.

`ActualParameterList` matches the parenthesized list of parameters to a method call.

`Block` matches a block of statements surrounded by curly braces.

`Class` matches a Java class declaration, including its body.

`Comment` matches a Java comment, both `//` style and `/*..*/` style.

`Constant` matches a constant expression, such as a number, character or string literal, or the identifier `null`.

`Expression` matches an expression.

`Field` matches a variable declaration in a class.

`FormalParameter` matches the declaration of a method parameter.

`FormalParameterList` matches the parenthesized list of parameter declarations in a method.

`Identifier` matches a user-defined identifier, such as a variable name, method name, or class name.

`Import` matches an `import` statement.

`Interface` matches a Java interface declaration, including its body.

`LocalVariable` matches a local variable declaration inside a method.

`Method` matches a method declaration, including its body.

`MethodBody` matches the body of a method, surrounded by curly braces.

`MethodCall` matches an expression that calls a method.

```
    MethodName is Identifier just before FormalParameterList
    MethodBody is Block ending Method
                         ignore nothing
    MethodCall is Identifier then ActualParameterList
```

`Statement` matches a statement.

`Type` matches a type, such as a class name or primitive type.

`VariableName` matches the name of the variable in a variable declaration.

```
    VariableName is Identifier in either field
                                  or localvariable
                                  or formalparameter
                               not in type
                               not in expression
```

A.7 Layout

Identifiers in the `Layout` namespace are defined by TC patterns, some in the file `Layout.tc`, and others in the file `HTML.tc`.

Delimiters

`CurlyBraces` matches a balanced set of curly braces, {...}.

`Parentheses` matches a balanced set of parentheses, (...).

`SquareBrackets` matches a balanced set of square brackets, [...].

```
            Parentheses is balances "(" with ")"
            SquareBrackets is balances "[" with "]"
            CurlyBraces is balances "{" with "}"
```

Form

`Form`	matches an HTML form.
`Control`	matches an HTML form control, such as a button or text field.
`Button`	matches an HTML form button.
`Checkbox`	matches an HTML checkbox.
`Menu`	matches an HTML drop-down menu.
`RadioButton`	matches an HTML radio button.
`Textbox`	matches an HTML text field.

```
Form is [form]
Control is either [input] or [textarea] or [select]
view source
  Textbox
   is either [input]
                 contains type-attr
                            contains either "text"
                                         or "password"
          or [input] not containing type-attr
          or [textarea]
  Button
    is [input]
         contains type-attr
                  contains either "button"
                               or "submit"
                               or "reset"
                               or "image"
  Menu is [select]
  Checkbox
      is [input]
            contains type-attr contains "checkbox"
  RadioButton
      is [input] contains type-attr contains "radio"
```

Image

`Image`	matches an image in a web page

```
Image is [img]
```

Line

`Line` matches a line in a plain text file.

`BlankLine` matches a line with nothing but whitespace characters in it.

`Break` is a synonym for `Linebreak`.

```
Line is nonempty from either start of page
                          or end of linebreak
                   to either linebreak
                          or end of page
Break is Linebreak
BlankLine is Line not containing Token
```

List

`List` matches an HTML list.

`Item` matches a single item in a list.

`BulletList` matches a bulleted list.

`Bullet` matches a single item in a bulleted list.

`NumberedList` matches a numbered list.

`NumberedItem` matches a single item in a numbered list.

`DefinitionList` matches an HTML definition list.

`Term` matches a term in a definition list.

`Definition` matches a definition in a definition list.

```
List is either [ul]
            or [ol]
            or [dl]
Item is either [li]
            or [dt] then [dd]
BulletList is [ul]
Bullet is [li] in BulletList
NumberedList is [ol]
NumberedItem is [li] in NumberedList
DefinitionList is [dl]
Term is [dt]
Definition is [dd]
```

Page

`Page` matches the entire page or file.

```
    Page is all
```

Paragraph

`Paragraph` matches (in plain text) a group of lines separated by blank lines, or (in HTML) a block-level element, such as `[p]` or `[li]`. Paragraph is defined by the Java class `lapis.parsers.SystemParser`.

Rule

`Rule` matches a horizontal rule (`<hr>` element in HTML).

```
    Rule is [hr]
```

Table

`Table` matches an HTML table.

`Row` matches a row in a table.

`Cell` matches a cell in a row.

```
    Table is [table]
    Row is [tr]
    Cell is either [td] or [th]
```

A.8 Style

Identifiers in the `Style` namespace are defined by TC patterns found in the file `HTML.tc`.

`Bold` matches text in boldface.

`Heading` matches a heading.

`Heading`*`N`* matches headings of size *`N`*, where *`N`* can range from 1-7.

`Italic` matches text in italics.

`Italics` is a synonym for `Italic`.

`Link` matches a hyperlink.

`Target` matches a hyperlink target (an `<a name=>` element in HTML).

`Underlined` matches underlined text.

```
Bold is either [b]
            or [strong]
Italic is either [i]
              or [em]
Italics is Italic
Underlined is [u]

Link is [a] starts <a> contains href-attr
Target is [a] starts <a> contains name-attr

Heading is either [h1]
               or [h2]
               or [h3]
               or [h4]
               or [h5]
               or [h6]
               or [h7]

Heading1 is [h1]
Heading2 is [h2]
Heading3 is [h3]
Heading4 is [h4]
Heading5 is [h5]
Heading6 is [h6]
Heading7 is [h7]
```

Appendix B

TC Pattern Operators

This appendix presents an alphabetical list of operators in the TC pattern language. For more details about the syntax and use of the pattern language, see Chapter 6.

B.1 And

```
expr1 and expr2
```

Matches regions that match both expressions. Equivalent to algebra operator $\cap$ (Section 3.5.1).
See also Section 6.2.15.

B.2 Anywhere After

```
anywhere after expr
```

Matches regions anywhere after some match to `expr`. Equivalent to algebra operator *after* (Section 3.5.2).

B.3 Anywhere Before

```
anywhere before expr
```

Matches regions anywhere before some match to `expr`. Equivalent to algebra operator *before* (Section 3.5.2).

B.4 Balanced from-to

```
balanced from expr1 to expr2
```

Matches a nested set of regions whose start delimiters match `expr1` and end delimiters match `expr2`. Equivalent to algebra operator *balances* (Section 3.6.8).

B.5 Case Sensitive

```
case sensitive expr
```

Forces literal and regular expression matches inside `expr` to be case-sensitive.

```
not case sensitive expr
```

Forces literal and regular expression matches inside `expr` to be case-insensitive (the default).

See also Section 6.2.8.

B.6 Contains

```
contains expr
```

Matches regions that contain at least one match to `expr`. Equivalent to algebra operator *contains* (Section 3.5.2).

B.7 Either-Or

```
either expr1 or expr2
```

Matches regions that match either `expr1` or `expr2`. The `either` is optional. Equivalent to algebra operator $\cup$ (Section 6.2.15).

See also Section 6.2.15.

B.8 End of

```
end of expr
```

Matches the end points of regions matching `expr`. Always returns a set of zero-length regions. Equivalent to algebra operator *end-of* (Section 3.6.1).

B.9 Ends

```
ends expr
```

Matches regions that end at the same point as `expr`. Ignores background regions around the end point. Equivalent to algebra operator ends_W (Section 3.6.13).

B.10 Equals

```
equals expr
```

Matches regions that match `expr`. Ignores background regions around both start point and end point. Equivalent to algebra operator equals_W (Section 3.6.13).

B.11 Flatten

```
flatten expr
```

Flattens the regions that match `expr`, by combining nested and overlapping regions into a single region. Equivalent to algebra operator *flatten* (Section 3.6.12).

B.12 From-To

```
from expr1 to expr2
```

Matches a flat region set consisting of regions that start with a match to `expr1` and end with the next match to `expr2`. Equivalent to algebra operator *fromto* (Section 3.6.8).

B.13 Identifier

```
identifier
```

Matches the regions matched by the pattern bound to `identifier` in the pattern library.
See also Section 6.2.3.

B.14 Ignoring

```
expr1 ignoring expr2
```

Sets the background set to the regions matching `expr2`, then evaluates `expr1` and returns its matches.
See also Section 6.2.14.

B.15 In

```
in expr
```

Matches regions lie in some match to `expr`. Equivalent to algebra operator *in* (Section 3.5.2).

B.16 Is

```
identifier is expr
```

Assigns the pattern `expr` to `identifier` in the pattern library, and returns the matches to `expr`.
See also Section 6.2.5.

B.17 Just After

```
just after expr
```

Matches regions that lie after and adjacent to some match to `expr`. Ignores background regions when testing for adjacency. Equivalent to algebra operator $\textit{just-after}_W$ (Section 3.6.13).

B.18 Just Before

```
just before expr
```

Matches regions that lie before and adjacent to some match to `expr`. Ignores background regions when testing for adjacency. Equivalent to algebra operator $\textit{just-before}_W$ (Section 3.6.13).

B.19 Literal

```
"string"
'string'
```

Matches regions consisting of the literal characters `string`. Either single or double quotes may be used to delimit the string.

See also Section 6.2.8.

B.20 Melt

```
melt expr
```

Melts the regions that match `expr` by combining nested, overlapping, or adjacent regions into a single region. Equivalent to algebra operator *melt* (Section 3.6.12).

B.21 Nonzero

```
nonzero expr
```

Matches regions that match `expr` and contain at least one character. Equivalent to algebra operator *nonzero* (Section 3.6.4).

B.22 Not

```
expr1 not expr2
```

Matches regions matching `expr1` that do not match `expr2`. Equivalent to algebra operator $-$ (Section 6.2.15).

See also Section 6.2.15.

B.23 Nth

```
nth expr
nth expr1 in expr2
nth expr1 before expr2
nth expr1 after expr2
```

The first form matches the nth region in the document that matches `expr`. The other forms match the nth match to `expr1` that lies in, before, or after each match to `expr2`.

The "`nth`" can be written in a variety of ways:

- `1st`, `2nd`, `3rd`, `4th`, ...
- `first`, `second`, `third`, ..., `tenth`
- `last, 2nd from last`, `3rd from last`, ...
- `second from last`, `third from last`, ...

Equivalent to algebra operator nth_n (Section 3.6.5).

B.24 Or

```
expr1 or expr2
```

Matches regions that match either `expr1` or `expr2`. Equivalent to algebra operator $\cup$ (Section 6.2.15).

See also Section 6.2.15.

B.25 Overlaps

```
overlaps expr
```

Matches regions that overlap some region matching `expr`. Equivalent to algebra operator *overlaps* (Section 3.6.3).

B.26 Overlaps End Of

```
overlaps end of expr
```

Matches regions that overlap the end point of some region matching `expr`. Equivalent to algebra operator *overlaps-end* (Section 3.5.2).

B.27 Overlaps Start Of

```
overlaps start of expr
```

Matches regions that overlap the start point of some region matching `expr`. Equivalent to algebra operator *overlaps-start* (Section 3.5.2).

B.28 Prefix

```
prefix identifier expr
```

Changes the current namespace to `identifier` for the scope of `expr`.
See also 6.2.7.

B.29 Regular Expression

```
/regexp/
```

Matches regions that match the regular expression `regexp`.
See Section 6.2.10 for the regular expression operators supported by LAPIS.

B.30 Start of

```
start of expr
```

Matches the start points of regions matching `expr`. Equivalent to algebra operator *start-of* (Section 3.6.1).

B.31 Starts

```
starts expr
```

Matches regions that start at the same point as `expr`. Ignores background regions around the start point. Equivalent to algebra operator starts_W (Section 3.6.13).

B.32 Then

```
expr1 then expr2
```

Matches regions that are the concatenation of a region matching `expr1` with a region matching `expr2` that lies after and adjacent to it. Ignores background regions when determining whether `expr1` and `expr2` are adjacent. Equivalent to algebra operator then_W (Section 3.6.13).

B.33 Trim

```
expr1 trim expr2
```

Matches regions that match `expr1` with an overlapping match to `expr2` removed from the start or end point. Equivalent to algebra operator *trim* (Section 3.6.14).

B.34 View

```
view source expr
view rendered expr
```

Forces literals and regular expressions inside `expr` to be matched against the HTML source or the rendered view of a web page. Has no effect on a plain text document.

See also Section 6.2.11.

Appendix C

LAPIS Commands

This appendix lists the script commands recognized by LAPIS, in alphabetical order. Standard Tcl commands are not included in this appendix; only new commands defined by LAPIS. For more information about LAPIS scripting, see Chapter 8.

C.1 Back

```
back [n]
```

Backs up to the previous document in the page history. The optional argument `n` is the number of pages to back up. This command has the same effect as the Back toolbar button.

See also: Section 8.8.

C.2 Calc

```
calc pattern
     [-count]
     [-sum]
     [-average|-mean|-avg]
     [-min]
     [-max]
     [-stddev]
```

Calculates statistics on the regions matching *`pattern`*. Only numeric regions are included in the statistics; nonnumeric regions are ignored. The statistics returned depend on which options are given:

- `-count` returns the number of numeric regions matching the pattern
- `-sum` returns the sum
- `-average` returns the mean of the regions. `-mean` and `-avg` are synonyms.

- `-min` returns the minimum of the matching regions
- `-max` returns the maximum
- `-stddev` returns the standard deviation

If only one option is given, then `calc` returns only the computed value. If multiple options are given, then `calc` returns a Tcl list of values in the order the options were given. If no options are given, `calc` computes all the statistics and returns a formatted display.

See also: Section 8.1.5.

C.3 Click

```
click pattern
```

Clicks on the hyperlink or form control described by *`pattern`*. Throws a Tcl exception if *`pattern`* does not match exactly one hyperlink or form control.

See also: Section 8.10.

C.4 Count

```
count pattern
```

Returns the number of matches to *`pattern`*.

C.5 Delete

```
delete pattern
```

Deletes all regions matching *`pattern`*. Synonym for `omit`.

See also: Section 8.1.3.

C.6 Doc

```
doc [string]
```

Returns the current document

```
doc string [-type type]
```

Sets the current document to *`string`*. With `-type`, the document is created with content type *`type`*. Possible content types are `text` and `html`. Without this argument, the content type is guessed from the content of *`string`*, defaulting to `text` if no valid HTML tags are found.

See also: Section 8.12.

C.7 Enter

```
enter pattern value
```

Sets all form fields matching *`pattern`* to the value *`value`*.

- For text fields, *`value`* is a string which is entered in the field.
- For menus and lists, *`value`* is the name of the selected value.
- For radio buttons and checkboxes, *`value`* should be one of the following: `on`, `off`, `yes`, `no`, `true`, `false`, `0`, `1`.

See also: Section 8.10.

C.8 Exec:

```
exec:command
```

Runs *`command`* as an external program.
See also: Section 8.6.

C.9 Extract

```
extract pattern
        [-startswith start]
        [-endswith end]
        [-separatedby sep]
        [-as type]
```

Extracts all regions matching *`pattern`*.

- `-startswith` prints *`start`* before each extracted region.
- `-endswith` prints *`end`* after each extraction region.
- `-separatedby` prints *`sep`* between each pair of regions (after the previous region's *`end`* and before the next region's *`start`*).
- `-as` converts the extracted regions to *`type`*, which may be either `text` or `html`.

See also: Section 8.1.1.

C.10 File:

```
file:filename
```

Loads the file named *`filename`* and returns it as the current document.
See also: Section 8.3.

C.11 Forward

```
forward [n]
```

Goes forward to the next document in the page history. The optional argument `n` is the number of pages to go forward. This command has the same effect as the Forward toolbar button.
See also: Section 8.8.

C.12 Ftp:

ftp:*//hostname/pathname*

Retrieves a file by FTP and returns it as the current document.
See also: Section 8.3.

C.13 History

```
history
```

Prints the page history to standard output. This command is designed for the LAPIS typescript shell (`lapis -tty`), which would otherwise have no other way to show the history.
See also: Section 8.8.

C.14 Http:

http:*//hostname/pathname*

Retrieves a web page by HTTP and returns it as the current document.
See also: Section 8.3.

C.15 Insert

insert *pattern string*

Inserts *string* at all points matched by *pattern*. Synonym for `replace`.
See also: Section 8.1.4.

C.16 Keep

keep *pattern* [-outof *recordpattern*]

Keeps only records matching *pattern* and deletes the rest.

- `-outof` specifies a record set rather than inferring it from *pattern*.

See also: Section 8.1.3.

C.17 Omit

```
omit pattern
```

Deletes all regions matching `pattern`.
See also: Section 8.1.3.

C.18 Parse

```
parse parser
```

Binds the patterns described by `parser` into the pattern library. The `parser` can be one of three possibilities:

- a filename ending in `.tcl`, which is interpreted as a script of Tcl commands;
- a filename ending in `.tc`, which is interpreted as a file of TC patterns;
- the name of a Java class implementing `lapis.Parser`. The class is loaded, an instance is created, and its `bind` method is called.

See also: Section 8.6.

C.19 Property

```
property [-get] name
```

Returns the value of the property named `name` on the current document.

```
property -set name value
```

Sets the `name` property to `value`.

```
property -list
```

Returns a Tcl list of the property names defined on the current document.
See also: Section 8.13.

C.20 Relocate

```
relocate
   [-base url]
   [-override]
```

Adds the HTML element `<base href=url>` to the current document. The *`url`* is obtained as follows:

1. From the `-base` argument, if specified.
2. From the `base` property of the current document, if any.
3. From the `url` property of the current document, if any.

If none of these can be found, `relocate` makes no change to the current document. If the current document already has a `<base>` element, `relocate` does nothing unless the `-override` option forces it to replace the existing `<base>`.

See also: Section 8.13.

C.21 Replace

```
replace pattern template
```

Replaces all regions matching *`pattern`* with *`template`*. The template may include embedded pattern substitutions surrounded by curly braces.

See also: Section 8.1.4.

C.22 Save

```
save [filename]
     [-backup extension]
```

Saves the current document to *`filename`*, or to the file the current document was loaded from if no *`filename`* is specified.

If the file already exists, the old contents are backed up to *`filename`*~ by default. The `-backup` option changes the backup extension from ~ to *`extension`*. If *`extension`* is the empty string, backup is disabled.

C.23 Show

```
show [-brief] [-all]
```

Prints the content of the current document to standard output. This command is designed for use in the LAPIS typescript shell (`lapis -tty`) and for scripts run from the command line.

- `-brief` displays at most a fixed number of lines, half from the start of the document and half from the end. The number of lines displayed is controlled by the Tcl variable `lapis::displayLimit`, which defaults to 25. This is the default when `show` is called interactively.
- `-all` displays the entire document. This is the default when `show` is used in a script.

C.24 Sort

```
sort pattern
     [-by keypattern]
     [-order [reverse] dictionary|numeric|unicode|random]
```

Sorts the regions matching *`pattern`*.

- `-by` specifies a sort key in each record. If no `-by` option is specified, the entire record is used as a sort key.
- `-order` specifies the sort order. Default is `dictionary`.

Multiple sort keys may be specified with multiple `-by` and `-order` arguments.

See also: Section 8.1.2.

C.25 Submit

```
submit [-form formpattern]
       [-button buttonpattern]
```

Submits a web form in the current page.

- `-form` specifies the form to submit. If `-form` is omitted, the first form in the page is used.
- `-button` specifies the button that should be pressed to submit the form. If no `-button` argument is given, the first button of type `submit` is pressed.

See also: Section 8.10.

Bibliography

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[And73] Michael R. Anderberg. *Cluster Analysis for Applications*. Academic Press, 1973.

[App02] Hypercard 2.4.1. http://www.apple.com/hypercard/, 2002.

[AS95] Eric Z. Ayers and John T. Stasko. In *Proceedings of the 4th International World Wide Web Conference (WWW4)*, pages 259–270, 1995.

[Aut02] Autocad. http://www.autodesk.com/, 2002.

[Bad99] Greg J. Badros. JavaML: A markup language for java source code. In *Ninth International World Wide Web Conference*, 1999.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[BL84] Vic Barnett and Toby Lewis. *Outliers in Statistical Data*. Wiley, 2nd edition, 1984.

[Bla01] Alan F. Blackwell. Swyn: A visual representation for regular expressions. In *Your Wish is My Command: Giving Users the Power to Instruct Their Software*. Morgan Kauffman, 2001.

[BLLJ98] Bert Bos, Hakon Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2 (CSS2) specification. Technical Report http://www.w3.org/TR/REC-CSS2/, W3C, 1998.

[BM80] Jon L. Bentley and Hermann A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13(2):155–168, 1980.

[Bru97] Amy Bruckman. *MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids*. PhD thesis, Masschusetts Institute of Technology Media Lab, May 1997.

[BYN96] Ricardo A. Baeza-Yates and Gonzalo Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, 1996.

[Car72] Lewis Carroll. The Jabberwocky. *Through the Looking-Glass and What Alice Found There*, 1872.

[CC97] Charles L. A. Clarke and Gordon V. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, May 1997.

[CCB95] Charles L. A. Clarke, Gordon V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, May 1995.

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[Cov96] Robin Cover. Dsssl – document style semantics and specification language. Technical Report 10179:1996, ISO/IEC, 1996.

[Cre97] Roger F. Crew. ASTLOG: a language for examing abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.

[Cyp93] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205–218. MIT Press, 1993.

[Dan92] Dan R. Olsen, Jr. Bookmarks: An enhanced scroll bar. *ACM Transactions on Graphics*, 11(3):291–295, July 1992.

[DAW98] Anind K. Dey, Gregory A. Abowd, and Andrew Wood. CyberDesk: a framework for providing self-integrating ubiquitous software services. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI '98)*, pages 47–54, 1998.

[DE95] Chris DiGiano and Mike Eisenberg. In *Symposium on Designing Interactive Systems (DIS '95)*, pages 189–197, 1995.

[DeJ98] Jacl and Tcl Blend. http://www.scriptics.com/software/java, 1998.

[DMO01] Steve DeRose, Eve Maler, and David Orchard. Xml linking language (xlink) version 1.0. Technical Report http://www.w3.org/TR/xlink/, W3C, 2001.

[Ede80] Herbert Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Rep. F59, Univ. Graz, Institute fur Informationsverarbeitung, 1980.

[FB74] Raphael A. Finkel and Jon L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[FGK93] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: a modeling language for mathematical programming*. Duxbury Press, 1993.

[Fin80] Craig A. Finseth. Theory and practice of text editors, or, a cookbook for an EMACS. Technical Memo 165, MIT Lab for Computer Science, May 1980.

[Fre98] Dayne Freitag. *Machine Learning for Information Extraction in Informal Domains*. PhD thesis, Computer Science Department, Carnegie Mellon University, November 1998.

[Fuj98] Yuzo Fujishima. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI '98)*, pages 101–108, 1998.

[GAM96] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings 4th Workshop on Program Comprehension*, pages 144–153, 1996.

[GDCG90] Sharon L. Greene, Susan J. Devlin, Philip Cannata, and Louis M. Gomez. No IFs, ANDs, or ORs: a study of database querying. *International Journal of Man-Machine Studies*, 32(3):303–326, 1990.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, 1983.

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[GPP71] Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, 1971.

[GT87] Gaston H. Gonnet and Frank W. Tompa. Mind your grammar: a new approach to modelling text. In *Proceedings of the ACM Conference on Very Large Databases (VLDB '87)*, pages 339–345, 1987.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[Gut84] Antonin Guttman. R-tree: a dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[Han71] Wilfred J. Hansen. User engineering principles for interactive systems. In *AFIP Conference proceedings, Fall joint computer conference*, pages 523–532, 1971.

[HH92] William C. Hill and James D. Hollan. Edit wear and read wear. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI '92)*, pages 3–9, 1992.

[HN83] Klaus Hinrichs and Jürg Nievergelt. In *Proceedings of the WG'83 International Workshop on Graph-theoretic Concepts in Computer Science*, pages 100–113, 1983.

[HN86] Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, December 1986.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17(5):935–938, 1988.

[Jak99] Jakarta Project. http://jakarta.apache.org/regexp/, 1999.

[Jav00] JavaCC. http://www.webgain.com/products/java_cc/, 2000.

[JB97] David Jackson and Michael A. Bell. String-pattern matching in a visual programming language. *Journal of Visual Languages and Computing*, 8(5-6):545–561, 1997.

[JK96] Jani Jaakkola and Pekka Kilpelainen. Using sgrep for querying structured text files. Report C-1996-83, University of Helsinki, Department of Computer Science, 1996.

[Joh75] Stephen C. Johnson. Computing Science Tech. Rep. 32, AT&T Bell Labs, Murray Hill, NJ, 1975.

[Kay01] Michael Kay. Xsl transformations (xslt) version 2.0. Technical Report http://www.w3.org/TR/xslt20/, W3C, 2001.

[KDE02] KDE desktop environment. http://www.kde.org/, 2002.

[KK94] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[KM93] Pekka Kilpelainen and Heikki Mannila. Retrieval from hierarchical texts by partial patterns. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, 1993.

[KM98] Thomas Kistler and Hannes Marais. WebL – a programming language for the web. In *Proceedings of the 7th International World Wide Web Conference (WWW7)*, 1998.

[KN98] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, pages 392–403, 1998.

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 2nd edition, 1988.

[Kru97] Bruce Krulwich. Automating the internet: Agents as user surrogates. *IEEE Internet Computing*, 1(4):34–38, 1997.

[Kul97] Zenon Kulpa. Diagrammatic representation for a space of intervals. *Machine Graphics and Vision*, 6(1):5–24, 1997.

[KWD97] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[Les75] Michael E. Lesk. Lex – a lexical analyzer generator. Computing Science Tech. Rep. 39, AT&T Bell Labs, Murray Hill, NJ, 1975.

[LH95] Jurgen Landauer and Masahito Hirakawa. Visual AWK: a model for text processing by demonstration. In *Proceedings of the 11th International IEEE Symposium on Visual Languages '95*, pages 267–274, 1995.

[Lis81] Barbara Liskov. *CLU Reference Manual*. Springer-Verlag, 1981.

[LNW98] Henry Lieberman, Bonnie A. Nardi, and David Wright. Grammex: Defining grammars by example. In *Proceedings of Human Factors in Computing Systems (CHI 98)*, pages 11–12, 1998.

[Lut00] Mark Lutton. Report on hacker altering mit grades: Not! *comp.risks*, 20(84), March 2000.

[LW77] Der-Tsai Lee and C.K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[LWDW01] Tessa Lau, Steven Wolfman, Pedro Domingos, and Daniel S. Weld. Learning repetitive text-editing procedures with SMARTedit. In Henry Lieberman, editor, *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, pages 209–226. Morgan Kaufmann, 2001.

[Lyx02] Lyx – the document processor. http://www.lyx.org/, 2002.

[Mac91] Ian MacLeod. A query language for retrieving information from hierarchic text structures. *The Computer Journal*, 34(3):254–264, 1991.

[Mau94] David Maulsby. *Instructible Agents*. PhD thesis, Department of Computer Science, University of Calgary, 1994.

[MB98a] James R. Miller and Thomas Bonura. From documents to objects: An overview of LiveDoc. *SIGCHI Bulletin*, 30(2):53–58, 1998.

[MB98b] Robert C. Miller and Krishna Bharat. SPHINX: a framework for creating personal, site-specific web crawlers. *Computer Networks and ISDN Systems*, 30(1–7):119–130, 1998.

[MC74] Robert Morris and Lorinda L. Cherry. Computer detection of typographical errors. Technical Report 18, Bell Laboratories, July 1974.

[McC81] Edward M. McCreight. Priority search trees. Tech Report CSL-81-5, Xerox PARC, 1981.

[Mic02] Microsoft. Complete tasks quickly with Smart Tags in Office XP. http://office.microsoft.com/assistance/2002/articles/oQuickSmartTags.aspx, 2002.

[Min92] Sten Minor. Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, 37(4):399–418, 1992.

[MM97] Robert C. Miller and Brad A. Myers. Creating dynamic world wide web pages by demonstration. Technical Report CMU-CS-97-131 (and CMU-HCII-97-101), CMU School of Computer Science, May 1997.

[MM99] Robert C. Miller and Brad A. Myers. Lightweight structured text processing. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 131–144, June 1999.

[MM00] Robert C. Miller and Brad A. Myers. Integrating a command shell into a web browser. In *USENIX 2000 Annual Technical Conference*, pages 171–182, June 2000.

[MM01a] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 161–174, June 2001.

[MM01b] Robert C. Miller and Brad A. Myers. Outlier finding: Focusing human attention on possible errors. In *Proceedings of User Interface Software and Technology (UIST 2001)*, November 2001. To appear.

[MM02] Robert C. Miller and Brad A. Myers. Multiple selections in smart text editing. In *Proceedings of the Sixth International Conference on Intelligent User Interfaces (IUI 2002)*, pages 103–110, 2002.

[MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, 1996.

[MP71] Robert McNaughton and Seymour Papert. *Counter-Free Automata*. MIT Press, 1971.

[MSC$^+$86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[Mye93] Brad A. Myers. Tourmaline: Text formatting by demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 309–322. MIT Press, 1993.

[Mye94] Brad A. Myers. User Interface Software Tools. http://www.cs.cmu.edu/~bam/toolnames.html, 1994.

[Mye98] Brad Myers. Natural programming: Project overview and proposal. Technical Report CMU-CS-98-101, Carnegie Mellon University School of Computer Science, January 1998.

[NBY95] Gonzalo Navarro and Ricardo A. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proceedings of the ACM Conference on Information Retrieval (SIGIR '95)*, pages 93–101, 1995.

[Nix85] Robert Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4):600–621, October 1985.

[NMW98] Bonnie A. Nardi, James R. Miller, and David J. Wright. Collaborative, programmable intelligent agents. *Communications of the ACM*, 41(3):96–104, 1998.

[Omn99] Omnimark. http://www.omnimark.com/, 1999.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Pap91] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 3rd edition, 1991.

[Pik87] Rob Pike. The text editor sam. *Software Practice and Experience*, 17(11):813–845, 1987.

[Pik94] Rob Pike. Acme: a user interface for programmers. In *Proceedings of the USENIX 1994 Winter Technical Conference*, pages 223–234, 1994.

[PK97] Milind S. Pandit and Sameer Kalbag. The selection recognition agent: instant access to relevant information and operations. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI '97)*, pages 47–52, 1997.

[PM96] John F. Pane and Brad A. Myers. Usability issues in the design of novice programming systems. Technical Report CMU-CS-96-132, Carnegie Mellon School of Computer Science, August 1996.

[PM00] John F. Pane and Brad A. Myers. Tabular and textual methods for selecting objects from a group. In *Proceedings of VL 2000: IEEE International Symposium on Visual Languages*, pages 157–164, September 2000.

[PRM01] John F. Pane, Chotirat Ratanamahatana, and Brad A. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2):237–264, February 2001.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, 1985.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Rit86] Jean-Francois Rit. Propagating temporal constraints for scheduling. In *Proceedings of the Fifth National Conference on AI (AAAI-86)*, pages 383–388. Morgan Kaufmann, 1986.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[Sch99] Gary L. Schaps. Compiler construction with ANTLR and java. *Dr. Dobb's Journal*, March 1999.

[SK98] Atsushi Sugiura and Yoshiyuki Koseki. Internet Scrapbook: Automating web browsing tasks by demonstration. In *Proceedings of User Interface Software and Technology (UIST 98)*, pages 9–18, 1998.

[ST92] Airi Salminen and Frank W. Tompa. PAT expressions: an algebra for text search. Report OED-92-02, University of Waterloo Centre for the New Oxford English Dictionary and Text Research, 1992.

[Sta81] Richard M. Stallman. In *ACM SIGPLAN SIGOA Symposium of Text Manipulation*, pages 147–156, 1981.

[TRH81] Tim Teitelbaum, Thomas Reps, and Susan Horwitz. The why and wherefore of the cornell program synthesizer. In *Proc ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 8–16, 1981.

[VGB92] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interface for language-based editing systems. *International Journal of Man-Machine Studies*, 37(4):431–466, 1992.

[W3C00] W3C. Extensible markup language (XML) 1.0. http://www.w3.org/TR/2000/REC-xml-20001006, October 2000. second edition.

[Wat82] Richard C. Waters. *SIGPLAN Notices*, 17(7):39–46, 1982.

[WCS96] Larry Wall, Tom Christensen, and Randal L. Schwartz. *Programming Perl*. O'Reilly, 2nd edition, 1996.

[WM92] Sun Wu and Udi Manber. Agrep – a fast approximate pattern searching tool. In *Proceedings of the Winter USENIX Technical Conference*, pages 153–162, 1992.

[WM93] Ian H. Witten and Dan Mo. TELS: Learning text editing tasks from examples. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 183–204. MIT Press, 1993.

[Woo81] Steven R. Wood. Z — the 95% program editor. In *Proc ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 1–7, 1981.